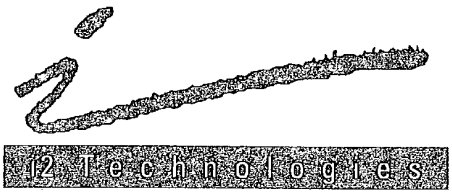
Intelligent Global
Supply Chain Management

Rhythm®
Supply Chain Planner
User Manual

i2 Technologies

The Intelligent Solution for
Global Supply Chain Management



The Intelligent Solution for Global Supply Chain Management

Rhythm®
Supply Chain Planner
User Manual

This manual was written, illustrated, and produced with the X/Motif and Windows NT FrameMaker document publishing software on a Sun SPARCstation IPC and Toshiba 400CDT, respectively.

Written and edited by Steven Chaples with contributions from the development and consulting groups of i2 Technologies, Inc.

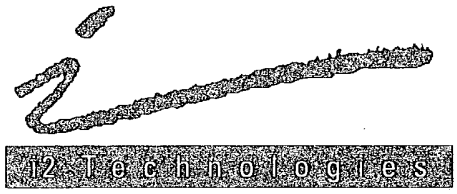
Version 3.04

i2 TECHNOLOGIES, INC.
909 East Las Colinas Blvd.
16th Floor
Irving, Texas 75039
USA

October 30, 1996

Revision History

Edition	Date	Reason for Revision
3.04 Beta	09/16/96	Beta Release
3.04A	10/14/96	Production Release



The Intelligent Solution for
Global Supply Chain Management

The information and/or drawings set forth in this document are undergoing continuous improvement. Cosmetic items are being enhanced to provide better visual appearance. Text is being reworded and added to improve the clarity of the information and to increase the ease of finding the information required.

If you have any comments or suggestions concerning this document, please write them and hand to your i2 Technologies representative, or mail to the address provided.

MANUAL NAME _____

COMMENTS_____

i2 TECHNOLOGIES, INC.
909 East Las Colinas Blvd.
16th Floor
Irving, Texas 75039
USA

October 30, 1996

Rhythm logo: i2 Technologies provides solutions for intelligent planning and scheduling. It simultaneously considers all constraints. The graphic shows a customer handling all the constraints simultaneously to obtain significant business results. The constraints are (counterclockwise): Due Date Planning (calendar), Throughput (drum encircled by arrow), Operating Expenses (dollar sign), Inventory (pallet with boxes), Lead Time (clock). Decisions are made using global rather than local criteria, although global information is filtered and combined with local information. Designed by Julie Arata.

Table of Contents

List of Figures

List of Tables

Section 1 Introduction

i2 provides the most flexible user interface available in any application. i2's Interactive Report Mechanism allows all user interfaces or Reports (including GUI windows, paper reports, and import / export files) to be easily customized. This means that there is only one mechanism to learn for customizing all user interfaces. The Interactive Report Mechanism allows the user extreme flexibility in how data is organized and displayed. All screens and paper reports provide spreadsheet-like multidimensional structured tables and charts (Bar, Line, Gantt, Pie, etc.) This flexibility even extends to how an individual data item is displayed and/or edited. The Interactive Report Mechanism provides access to anything in the Model Reference depending on the security level of the user. This tool provides excellent support for custom analyses generated by the user themselves

This document describes the graphical user interface that is used to create worksheets, layouts, and reports. It describes the elements that are available for computing data to be displayed and the elements that are available for displaying this data once it has been computed.

The phrases *Rhythm® Worksheet* and *Interactive Report* are sometimes used synonymously. When used in this way, they refer to the mechanism that allows users to write their own reports which specify both the data and the format of that data in a GUI window. However, the term worksheet, strictly speaking, has a more limited definition.

1.1 Mission Statement

i2 Technologies' mission is to enhance the clients' competitive position by providing planning and scheduling tools which will enable very responsive global supply chain management at a low cost. Client companies are facing a significant number of supply chain problems that continue to escalate:

- Retailers transferring logistics responsibility more to manufacturers with concepts like Vendor Managed Inventories, etc.
- Increasing emphasis on 100% service level
- Shortening product life cycles and accelerated obsolescence
- Engineering Change Requests (ECR)
- Promotions
- Transportation is becoming a significant part of the cost in many businesses
- Packaging
- Consolidation and Cross-Docking
- Shelf life of products

1.2 Purpose

The *Rhythm*® Interactive Reports are a new mechanism for generating reports that is designed to encompass report generation, GUI window (interactive report) generation, and data file generation. It exceeds the capabilities of standard spreadsheet programs, so is not just another spreadsheet program.

The *Rhythm*® Interactive Reports are a response to requests for greater flexibility and easier customization by users. It satisfies the following needs:

- It is designed to meld together *Rhythm*™'s file generation functionality which is used for data/spec file definition and *Rhythm*™'s export mechanism which is used for definition of what can be put in GUI windows. The result is an integrated specification which defines both what can go in windows and what can go in data files. Anything that can be loaded through data files can also be edited in GUI windows. The *Rhythm*® Model Reference defines what can be accessed from or set into the internal software mechanisms from either GUI windows or data files.
- It unifies the following three activities from the user point-of-view.
  - Generating a data file for import or export is done by defining a report.
  - Generating a traditional printed report is done by defining a report.
  - Generating a GUI window is done by defining a report (an interactive report). The user needs to learn only one mechanism to cover all activities.
- It was designed to be more programmable and customizable than *Rhythm*™. The programmability is great enough that developers define all of the GUI windows using the mechanism. Thus, everything that is put into a window is modifiable by users. Users are able to define almost any window that developers define.
- It was designed to be programmable by users. The **.ad* files are an example of an interface not suitable for users. C++ is also not suitable. A language with a great deal of power and yet easily understood and utilized by personnel, planners, sales people, and CEOs is needed. By far, the most successful and widely used programming paradigm is the spreadsheet. The spreadsheet is currently heavily used by all of the people just mentioned. Most of the target users are familiar with spreadsheets. The programmability and customization mechanisms have been modelled after the spreadsheet.

1.3 Description

Much of the interface looks like traditional spreadsheets. Users need to feel comfortable in the interface, and need to feel like they know how to program the interface with only a small introduction. Also, a tabular, spreadsheet-like format is one of the most useful formats for displaying data. It is no accident that the format was chosen by 123, VisiCalc and early spreadsheet pioneers. Note that the dominant manufacturing software is not an MRPII package but is the spreadsheet.

The *Rhythm*® Interactive Reports are designed so that an interface may be prototyped and connected to the internal software mechanisms in a matter of minutes instead of

weeks. Developers do not need to be consulted and the interface is as much a part of the system as any other interface.

If a user likes how he saw data displayed elsewhere, he can call on *Rhythm*® to display a window according to his specifications, and to do it quickly. If spreadsheets can be written, then what was just done can be done for any window, report, or data file in the system.

In summary, *Rhythm*® is not a new spreadsheet program. It is a new GUI / report / data file mechanism that is designed to be as easy to program as a spreadsheet. The only thing that is proven to be as easy to program as a spreadsheet, is a spreadsheet-like mechanism. The only programming mechanism that is familiar to most target users is a spreadsheet-like mechanism. So, the new *Rhythm*® Interactive Reports mechanism is a spreadsheet-like mechanism (but it is not a new spreadsheet program).

1.4 Modules

The *Rhythm*® family of intelligent supply chain management products includes all of the functions that today's companies need to manage their supply chain to achieve significant financial performance and a powerful competitive edge. *Rhythm*® has been designed to evolve with users'continually changing requirements and the developing nature of advanced object-oriented and GUI technology.

The *Rhythm*® family includes the following stand-alone modules.

- Rhythm SCP (Supply Chain Planner)
- Rhythm OIL (Object Interaction Language)
- Rhythm RhythmLink

1.4.1 Rhythm

The *Rhythm*® module is extensible by design, and so has add-on packages. It is composed of the following application modules:

- Factory Planning
- Advanced Scheduling
- Distribution Planning
- Multi-Enterprise Communication
- Problem-Oriented Planning
- Strategy-Driven Planning
- Available-to-Promise

1.4.1.1 Factory Planning

Factory Planning is an intelligent superset of traditional MRP II and overcomes the shortcomings of traditional MRP II. Traditional MRP II focuses on requirements analysis, i.e., given demand, or a production plan, the focus is on translating it to a master production schedule (MPS), material requirements plan (MRP), and a capacity requirements plan (CRP). Factory Planning calculates requirements but can also plan in the

reverse direction. Given capacity, material or other constraints, Factory Planning can plan upstream and downstream of the constraint. Factory Planning generates a can-do quantity based on constraints and the calculation of an optimal MPS and factory operating plan. For example, if a user wants to analyze the impact of an order reaching a node/facility three days later than originally planned, Factory Planning can provide visibility to the implications of the delay and predict the downstream inventories, resource loads, service levels etc. The plans developed by Factory Planning are feasible to execute since all constraints are recognized and accommodated. The time and effort required to generate a plan is less than 10% of that required by traditional MRP II.

Factory Planning supports a multi-plant environment with two way inter-dependency.

1.4.1.2 Distribution Planning

i2's global supply chain manager is an intelligent super set of traditional DRP and overcomes the shortcomings of traditional DRP. Traditional DRP focuses on requirements analysis, i.e., given a demand signal, the focus is on translating this to demand on upstream locations. i2's global supply chain manager calculates requirements but can also plan in the reverse direction. Given capacity, material or other constraints, i2's global supply chain manager can plan upstream and downstream of the constraint. It can thus carry out can-do analysis based on the presence of a constraint. For example, if a user wants to analyze the impact of a part reaching a location three days later than originally planned, i2's global supply chain manager determines the full impact of the delay; predicts the downstream inventories, transportation loads, service levels etc. The plans developed are feasible to execute since all constraints are recognized and accommodated. Time and effort required to generate a plan is less than 10% of traditional DRP. i2's global supply chain manager can function, stand-alone, in the distribution domain or can be used to plan the entire supply chain. Distribution, manufacturing and material purchasing plans can be formed simultaneously. For example a customer requires fifteen line items to be shipped to a given location, the following decisions are made *simultaneously* (for synchronized and cost effective delivery):

- Transportation
- Sourcing (which warehouse? which factory?)
- Allocation (of material and capacity to demand over time)
- Manufacturing
- Purchasing

Intelligent route selection ensures the selection of an 'optimal' replenishment path consisting of the appropriate distribution center, warehouse, manufacturing facilities and transportation options. Optimal is what best meets the desired business objectives.

Traditionally, plans are generated, item by item, sequentially. i2's global supply chain manager plans all demand simultaneously e.g. 150 SKU's to be replenished at 10 locations. i2's global supply chain manager considers these 1500 replenishment requests *simultaneously*. Simultaneous planning allows intelligent consolidation, allocation, sourcing and transportation planning Lead times in the system are not assumed to be fixed. All lead times in the supply chain (transportation, manufacturing, etc.) are a result of constraint based planning and scheduling or chosen options. Full pegging of demand is supported throughout the entire supply chain. Visibility of source and priority of

demand is maintained as it is propagated upstream so appropriate trade-offs can be made. The consolidation routines support full truckloads (TL) and less than full truckloads (LTL) with full visibility to the complete supply chain. Each node, or resource, in the supply chain like a warehouse, transportation center, plant, etc. can have its own work calendar. Arrivals and deliveries are synchronized at the distribution centers (DC's) to facilitate cross-docking.

1.4.1.3 Advanced Scheduling

Advanced Scheduling adds the fancier detailed scheduling issues, such as intelligent ways of dealing with sequence dependent setup.

1.4.1.4 Strategy-Driven Planning

Strategy-Driven Planning allows users to define the automated scheduling strategy (such as cost optimization). Without that, the automated planner just does its own thing.

1.4.1.5 Available-to-Promise

Available-to-Promise is not just a traditional a ATP. It is an advanced, pre-allocated, capacity-constrained ATP.

1.4.1.6 Multi-Enterprise Communication

Multi-Enterprise Communication corresponds to Interplant. It provides inter-*Rhythm*-engine connection and automated Request-Promise capabilities.

1.4.2 Rhythm/OIL

Object Interaction Language (OIL) is the name of the interactive report editors (that constitute a 4GL) which allow users to design their own windows for interaction with *Rhythm*® (and other products). Without this language, users' ability to modify the interface would be limited.

1.4.3 RhythmLink

The RhythmLink product provides a clean interface to import and export data from various databases (e.g. Oracle, Sybase, Informix, VSAM, DB2). This is wrapped in integrated fashion into the import/export mechanism. SQL expertise is not needed for most situations.

1.5 Audience Description

This manual is directed at the production planners and schedulers and the Information Services personnel responsible for operating and maintaining the *Rhythm*® system. Typically, the production planners and schedulers interface with the *Rhythm*® system on a daily basis through the *Rhythm*® user interface software application, a hierarchical system of pull down menus and pop-up windows. They may access a *Rhythm*® server application locally, by accessing the *Rhythm*® server on the host machine via *Rhythm*® client application, or remotely, by accessing a *Rhythm*® server on another machine.

1.5.1 Role of the Production Planner and Scheduler

The production planner and scheduler is responsible for the following functions:

- Collect all data relevant to the particular scenario. For example:
  - bill of material data
  - demand order data
  - resource data
  - routing data
  - inventory data
  - vendor data
- Build files of the data for input to *Rhythm*®
- Create specification files that format the data files for input to *Rhythm*®
- Run the *Rhythm*® application software
- Use the *Rhythm*® Standard Reports to model the data
- Use the *Rhythm*® Interactive Report Editor to design worksheets, layouts, and reports for modeling data
- Generate graphs and reports from the *Rhythm*® application to highlight areas of concern in plans and schedules applied
- Re-plan and re-schedule based on new data

1.5.2 Role of the System Administrator

The System Administrator is responsible for the following functions:

- Determine system platform requirements
- Obtain and install system hardware and software
- Install *Rhythm*® software on the customer platform
- Install *Rhythm*® software updates
- Monitor performance of the system platform and of *Rhythm*®

The System Administrator may also be responsible for the following functions:

- Collect all data relevant to the particular scenario. For example:
  - bill of material data
  - demand order data
  - resource data
  - routing data

- inventory data
- vendor data

- Build files of the data for input to *Rhythm*®
- Create specification files that format the data files for input to *Rhythm*®

1.6 How to Use This Manual

The best way to learn *Rhythm*® the most quickly and to learn it efficiently is to follow these steps:

- Read the Introduction, Purpose, and Description
- Read the section on expression language
- Use the *Rhythm*® Standard Reports to model data
- Use the *Rhythm*® *Object Interaction Language* (*OIL*™) to design worksheets, layouts, and reports for modeling data

1.7 Project Nomenclature

AMD	Add, Modify, Delete
ATP	Available to Promise
BOM	Bill of Materials
CAO™	Constraint Anchored Optimization
COM	Consumer Oriented Manufacturing
COMMS	Consumer Oriented Manufacturing Management Systems
CRP	Capacity Requirements Planning
DRP	Distribution Requirements Planning
DS	Detailed Sequencing
DTS	Defect Tracking System
ERP	Enterprise Resource Planning
FGI	Finished Good Inventory
FP	Factory Planning
GUI	Graphical User Interface

JIT	Just In Time
MES	Manufacturing Execution Systems
MP	Master Planning Module
MPS	Master Production Scheduling
MRP	Material Requirements Planning
MRP II	Manufacturing Resource Planning
POP	Problem-Oriented Planning
SCP	Supply Chain Planning
SDP	Strategy-Driven Planning
SFC	Shop Floor Control
SFMP	Synchronous Flow Management Planner
UI	User Interface
UID	User Identification
UOM	Unit Of Measure
WIP	Work In Progress

1.8 References

1. APICS Dictionary
2. *Rhythm*® Supply Chain Planner Model Reference Manual
3. The UNIX Programming Environment
4. X Window System User's Guide

Section 2 Supply Chain

2.1 Introduction

i2 is a supplier of an integrated decision support system for global supply chain management. i2's global supply chain manager addresses the principle business drivers of Return on Assets, Delivery Performance, Profit Contribution, and Responsiveness (Order Lead Time) with product quality understood as a given. Using a totally new approach, i2 provides its customers a method to achieve significantly superior results in these key business drivers. This new approach does supply chain planning concurrently rather than sequentially. Distribution, transportation, sourcing, allocation, manufacturing, and procurement plans can be *simultaneously* formed. These plans recognize and accommodate all of the constraints in the chain: material, capacity, manpower, transportation, warehousing. The system also handles in real-time the bi-directional propagation of "changes". This bi-directional planning provides the ability to see problems and changes without the need to go back to the starting point of the planning process, make modifications and uni-directionally pass them back through the process. The uni-directional start-over process loses the cause and effect relationship that is necessary to effectively solve these unforeseen problems.

Another advantage of this approach is that it allows very rapid what-if analysis for large and complex problems. i2's global supply chain manager makes it very easy for planners to perform the following types of what-if analysis:

- When can a new product be shipped to a key customer? How many can be shipped by a certain date? When can a certain amount be shipped?
- What locations or channels should receive allocations of product?
- What mode of transportation and which carriers should be used? What should the routing be?
- Is capacity being allocated among key customers or channels in the most profitable way
- Which work centers are impacted the most by the new forecast?
- What is the overall effect on the supply chain due to the forecast change?
- Which orders will be impacted by a late component delivery from a supplier?
- What materials need to be expedited and how much overtime is required to meet a new and unplanned order?
- Which subassemblies or fabrication efforts can be delayed because an order has been put on hold for two weeks?

In traditional manufacturing systems, planners must go through several different applications taking hours or days to answer these questions. i2's global supply chain manager allows planners to answer these questions literally in seconds. i2's global supply chain manager is very flexible so that it can be used to answer these questions in a wide variety of environments. i2's global supply chain manager is designed to be used in multiple

distribution or plant situations as well as for planning the operations of individual facilities. It takes into consideration everything from customer's demands to supplier's capabilities. In a distributed, multi-company enterprise, companies for the first time can tie the entire supply chain into one real-time, concurrent planning environment!

2.2 Global Supply Chain Manager

2.2.1 Forecast Management

i2 provides an extremely effective forecast management tool. Forecasts can be received at various levels of abstraction. i2's global supply chain manager provides tools that aggregate and disaggregate forecasts at many different levels. Tools are also provided to manage forecast variances and their calculations.

2.2.2 ATP and Automated Order Promising

i2 provides real time order quotation based on the simultaneous consideration of material, capacity, transportation and other business constraints. The system provides the quantity that can be delivered by the requested date and the date by which the total requested quantity can be delivered. The ATP mechanism allows for the allocation of materials and capacity to channels. The consumption against allocations is continuously monitored. Allocation strategies intelligently adjust allocations to maximize achievement of business objectives.

2.2.3 Sourcing

Demand at a given location can be filled, sometimes, by alternate suppliers. They can be different manufacturing locations, stock locations or third party suppliers. Sourcing is the process of choosing the optimal supplier. Sourcing is an integral part of the distribution planning, factory planning and procurement planning parts of i2's global supply chain manager. i2's global supply chain manager's sourcing is based on cost based optimization. The following types of variables are considered in choosing between alternate sources:

- Transportation costs
- Processing costs
- Material and capacity availability and associated usage costs
- Service performance and associated costs

User defined sourcing algorithms can also be easily accommodated into the problem solving architecture.

2.2.4 Allocation

Allocation is the process of allocating materials and capacities to competing demand sources. This is particularly important when they are in short supply. Allocations can be made to many different channels. These channels can be based on geography, industry, products, product profit margins, or almost any other category. The allocation process works at different levels:

- end-item level: allocate finished goods available in one country to meet an urgent demand in another country? Allocate finished goods to a customer with a lower profit contribution or reserve it for customers with quick response requirements?
- intermediate and raw material level: The issues are similar to the end-item level with some differences. Materials should be allocated to demand only if all other materials and capacity, that will be required downstream, are available within a prescribed amount of time.
- resource level: When resources are overloaded then the available capacity needs to be allocated. Capacity should be allocated to demand only if all other materials and capacity that will be required downstream are available within a prescribed amount of time. There is no point in allocating capacity to demand that will not generate throughput; it only creates inventory. Capacity and materials can be pre-allocated or reserved for certain classes of demand. Examples of these classes are channels, geographies, customers, etc. i2's global supply chain manager tracks consumption against reservations and allocates based on availability.

2.2.5 Inventory Planning

i2's global supply chain manager has the ability to model different inventory policies at different nodes. For example:

- Min, max and trigger (re-order point)
- Days of supply (based on requirements)
- Service level (statistical)
- Aggregated level

i2's global supply chain manager is capable of planning the necessary response buffers for Assemble-To-Order and Package-To-Order businesses. It can also handle the planning of constraint buffers such that the constraints to throughput do not starve. It also supports pull based planning schemes such as JIT/Kanban. Inventory lot sizing, at a node, will be dependent on downstream constraints e.g. transportation mode, setups, etc.

2.2.6 Procurement and Outsourcing

i2's global supply chain manager supports inter-company and intra-company supply chains. Any outside supplier, even a customer, can be made a part of the supply chain and planned. Supplier capacities and other constraints can be modeled as part of the supply chain resulting in superior plans. Sourcing logic is used to make smarter sourcing decisions when multiple suppliers for the same parts exist. Decisions can be based on vendor capacities, costs, lead times etc.

2.2.7 Software Technology

i2 is using the very latest in object-oriented software technology. i2 has an exceptional understanding of this technology and knows that being object-oriented alone does not guarantee a world class software system. To make use of object-oriented technology it is necessary to have a deep knowledge of the problem that the technology is meant to address. i2 has put together a development team that is the industry leader in the plan-

ning and scheduling problem domain. Using this understanding of the planning and scheduling problem, i2 has created the Object Interaction Language (OIL) architecture. The Object Interaction Language architecture is an open, extensible architecture that avoids the problems of building a product around a generic object-oriented architecture.

2.2.8 Extensible Model Architecture

The Object Interaction Language framework is designed to allow customization to be easy, fast, and non-intrusive. i2 provides extensions which are pre-defined sets of additional fields that can give different, custom behavior to the entire model. Extensions (Extensible Model Architecture™)can be plugged together to make custom models based on users specific needs. User-defined fields can easily be added to each major model. A library of extensions is available from which a user can pick appropriate extensions for each element of the Supply Chain. New extensions can be created with custom behavior for those functions that are not included in the base system. Extensions can also be programmed by field settings, i.e. fields can contain Expressions, Expert System rules, Linear Programming equations, etc.

2.2.9 Sophisticated Planning Mechanism

i2's global supply chain manager allows for Problem-Oriented Planning™ through the generalized use of its proprietary Constraint Anchored Optimization™ algorithm. This sophisticated planning mechanism provides users the ability to define their own strategies for planning, decide their own goodness criteria, and how much auto-resolution they want. The system totally integrates Material and Capacity Planning, Master and Operational Planning, as well as Factory and Distribution Planning. The automated planning cooperates with any manual intervention desired by the planner. The sophisticated planning mechanism provides true what-if planning with full undo capability.

Section 3 Mechanics of Using TIPS

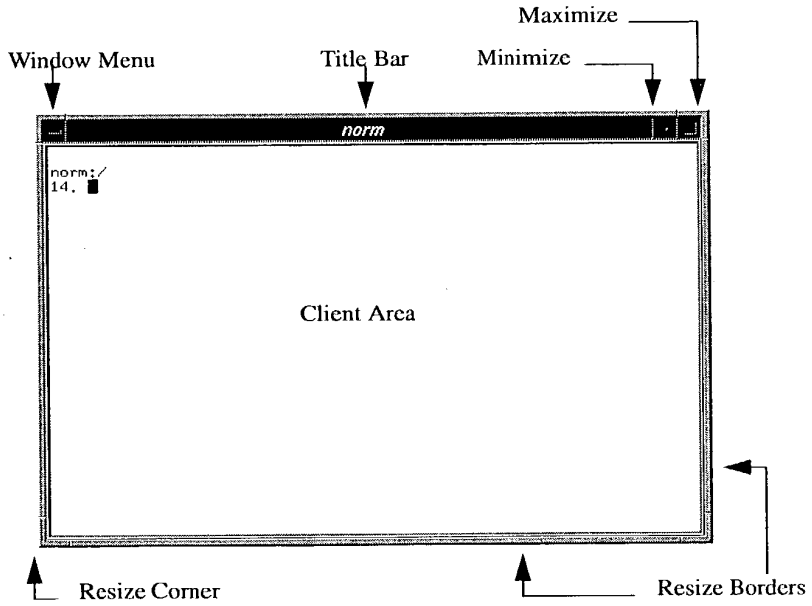
3.1 Introduction

This section describes the structure of screens that users will see while running *TIPS*. It describes the function of each symbol on a screen and within each menu.

3.2 Window Anatomy

The OSF/Motif Window Manager (MWM) provides the windows presented by the *Rhythm*® software. These windows contain various functional components accessible with the controls on a mouse. In general, a window consists of the components described in the following paragraphs. See FIGURE 1.

FIGURE 1 OSF/Motif Window Anatomy



3.2.1 Client Area

The client area is the portion of the window, inside of the window frame, used for performing *Rhythm*® functions. This area may be divided into multiple work areas (see *Routing*).

3.2.2 Resize Borders

The installation of the application determines the initial size of a window. The size may need to be adjusted for:

- personal preference
- easier to see
- better organization of all windows on the screen

A border frame surrounds the window. Each of the four borders of the frame may be used to resize the window by following these steps:

- move pointer to one of the four resize borders
- one of the ↔ should appear
- press and hold left mouse button
- move mouse, thus dragging the border, until that border is moved appropriately
- may move in two opposing, parallel directions

3.2.3 Resize Corners

The installation of the application determines the initial size of a window. The size may need to be adjusted for:

- personal preference
- easier to see
- better organization of all windows on the screen

A border frame surrounds the window. Each of the four corners of the frame may be used to resize the window by following these steps:

- move pointer to one of the four resize corners
- one of the ↘ should appear
- press and hold left mouse button
- move mouse, thus dragging corner, until that corner is moved appropriately
- may move in all directions

3.2.4 Title Bar

The title bar is the horizontal bar that exists between the window menu button and the window control buttons at the top of a window. The title in the title bar identifies the window. Pressing and holding the left mouse button while the pointer is in the title bar and dragging the mouse moves the window. Clicking the left mouse button while the pointer is in the title bar of a window partially obscured by other windows brings the window to the front of the stack of windows.

3.2.5 Window Control Buttons

The window control buttons are push buttons located in the upper right corner of a window:

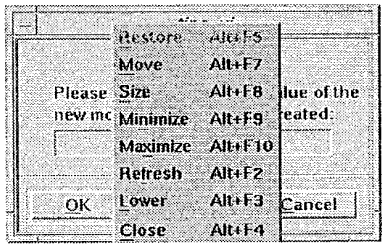
- Minimize Button - located to the immediate right of the title bar. Click this button to shrink the window to an icon at the bottom of the screen. This button serves the same function as the Minimize function in the window menu.
- Maximize Button - located to the far right of the title bar. Click this button to enlarge the window to a full screen. This button serves the same function as the Maximize function in the window menu.

3.2.6 Window Menu Button

The window menu button is a push button located to the immediate left of the title bar. Double clicking the pointer on this button closes the window. Clicking the pointer on this button (or pressing <*Shift*>+<*Esc*> on the keyboard) displays the window menu (See FIGURE 2):

- Restore - returns a window to its normal size after being minimized or maximized
- Move - changes the location of a window on the screen
- Size - stretches or shrinks a window in the direction indicated by the pointer
- Minimize - shrinks a window to an icon
- Maximize - enlarges a window to full screen
- Refresh - redraws the image of the window on the screen (important if the system corrupts the image)
- Lower - moves a window to the back of the stack of windows
- Close - closes a window

FIGURE 2 Window Menu Button



3.3 Cursor and Pointer Shapes

The shape of a cursor or pointer may change, depending on its position in a window. The change in shape indicates that the cursor or pointer has attained a different function.

3.3.1 Cursor Shapes

The following cursor shapes may be seen:

- I-bar - this shape indicates that text may be inserted
- | or ^ - insertion cursor
- ▮ or _ - overstrike cursor
- ▭ - location cursor

3.3.2 Pointer Shapes

The following pointer shapes may be seen:

- ↗ - standard pointer image. Shows the location at which the mouse is currently pointing
- Watch - indicates that the system is currently busy
- ↔ - an arrow pointing up, down, left, or right perpendicular to a short line tangent to the arrow at its point. This shape appears when the pointer is moved to one of the four resize borders of the active window. The window may then be resized in both directions that are parallel to the direction of the arrow by moving the mouse such that the arrow moves in one of those two directions.
- 4-Way Arrow - This shape indicates that a window may be moved. This shape appears when the left mouse button is pressed and held after the pointer is moved to the title bar of the window. The window may them be moved by moving the mouse while still holding the left mouse button.
- ⇲ - an arrow pointing diagonally into the bend in a right angle. This shape appears when the pointer is moved to one of the four resize corners of the active window. The window may then be resized from that corner by moving the mouse in any direction.
- + - this shape is used for sighting. Its most common use within *Rhythm®* is for resizing one partition of a *Rhythm®* window (see *Routing, Tasks Planned, Manual Load Balancing*):
  - sight the center of the box
  - press and hold the left mouse button
  - move the cross hair vertically to resize that portion of the window

3.4 Menus

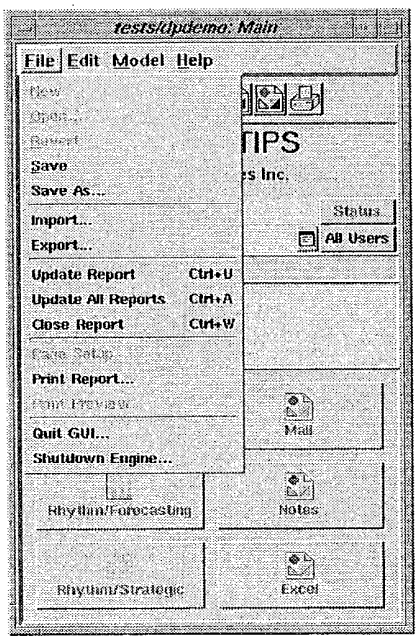
3.4.1 Anatomy

A menu is a list of selections that appear when the left mouse button is pressed and held on a menu object. See FIGURE 3. Examples of menu objects are:

- menu items appearing in an application's menu bar, appearing immediately below the title bar. Occasionally, a header area may appear between the title bar and the menu bar. The header area contains non-editable, informative text.
- window menu button

Within a menu list, some of the items may appear in a lighter type than the other items. A *de-emphasized* item indicates a function that is currently inactive. Also, within a pull-down menu list, some of the items may be followed by a keyboard sequence such as *Alt + F* or *Ctl + D*. These *accelerators* may be provided for frequently used functions, allowing execution of these functions from the keyboard without needing to display the menu. The disadvantage, though, is that the keyboard sequence must be remembered unless a keyboard template has been provided. Underlined letters in a menu bar also serve as accelerators for displaying the pull-down menu list by pressing the *<Alt>* key followed by the letter. An underlined letter in an option of a pull-down menu list serves as an accelerator by pressing the letter to access the option. See FIGURE 3.

FIGURE 3 Menu Anatomy



3.4.2 Pull-Down Menu

A pull-down menu is always associated with an application's menu bar, appearing immediately below the title bar. By pressing and holding the left mouse button on an item in the menu bar, a pull-down menu for that item will appear. While still holding the left mouse button, a function in the list may be selected by dragging the pointer to the function and releasing the mouse button. See FIGURE 3.

3.5 Dialog Boxes

A dialog box is a window that requests or displays information or instructions. In a menu system, one is usually heralded by an ellipsis immediately following the menu item which accesses the dialog box. For example:

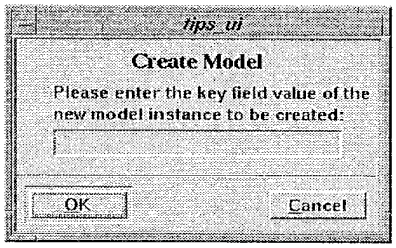
Save As...

3.5.1 Anatomy

A dialog box window will contain a text entry box, an information message, or a data area, often followed by push buttons that represent all possible actions for the text entered or all possible responses to the information displayed. See FIGURE 4.

A push button is selected by moving the pointer to the desired button, then clicking the left mouse button. The default push button selection is distinguished from the other push buttons by having a different border. The default may be selected by pressing <*Return*> on the keyboard. To move between push buttons, the <*Tab*> key on the keyboard may be pressed.

FIGURE 4 Dialog Box



Section 4

Import Files

4.1 Introduction

The *import file* specifies the contents of each data file to be read. The import file can be explained as *how to read or write the customer's file*. Thus, *Rhythm*® can be customized to read the data in a format that is easy for the customer to provide.

An import file is a worksheet form which specifies how to import data to *Rhythm*® from ASCII data files. It can be thought of as a one row worksheet, where the row is repeated once for every line (record) of the data file. This row consists of cells. All cells in the import file correspond to a single record in the data file (a data file record is a line of tab delimited fields ending with \n). Each cell in the import file corresponds to a single field in the datafile. Cells are named, so that intermediate values can be saved. Import files contain a file extension of *.imp*.

☞ Note: Keep in mind that a line in a data file can create multiple models.

This section contains the following topics.

Topic
Reading Import Files and Data Files
Import File Layout Formats
Import File Format
Parts of a Import File Format
Import File Expressions
Import File Examples

4.2 Procedure

The following procedure shows where import file writing fits within the entire process:

- Build data files (*.dat* files)
- Write import files (*.imp* files) for each data file. These files specify how the data files are read.
- Generate reports (*.rpt* files) - worksheets whose cells compute layouts
- Run *scp_engine*
- Run *scp_ui*

4.3 Reading Import Files and Data Files

Table 1 presents the process that *Rhythm*® uses for reading import files and data files.

Table 1: Reading Import Files and Data Files

Stage	Description
1	*scp_engine* is run
2	Command line options and option (*.opt*) files are read.
3	System import files are read: import files in the system directory (as specified by the *system* command line option and by option (*.opt*) files) are read one at a time
4	System data files are read: as each system import file is read, the data file(s) specified by the *file* command in each import file are read one record (line) at a time using the import file format
5	*scp_engine* import files are read: import files in the data directory (as specified by the *data* command line option and by option (*.opt*) files) are read one at a time
6	*sf_const* variables are set once at the beginning of file reading
7	*read* variables are set after each record is read (so the # variables are set), and before the various cell *set* expressions are evaluated (so the *read* variable can be used by those expressions)
8	The root model into which an import file reads is stored in the *this* variable.
9	All references to the same model are collected.
10	The groups of models are ordered so that dependent models can be created properly. Since a data file record may have fields which correspond to any number of different models and submodels, import files must handle complex groupings and orderings of models and fields.
11	*scp_engine* data files are read: as each *scp_engine* import file is read, the data file(s) specified by the *file* command in each import file are read one record (line) at a time.
12	The appropriate models are created based on the model groupings.
13	The models are initialized.
14	When all groups have been used, the next record is read.

4.4 Importing Data

4.4.1 Introduction

Data is imported at the time the *Rhythm*® engine is run. The engine options *data*, *system*, and *user_data* specify in what directories data files may be found. These options may be specified as command line options or read into the *User* model in the *Rhythm*® Model Reference Manual from a *user.dat* file via a *user.imp* import file. See User Customization for all available *Rhythm*® engine and *Rhythm*® UI options:

- *include* - specifies the pathname to the directories that contain data for both the *Rhythm*® engine and UI. Will usually be '.' (current directory, i.e. directory from which *scp_engine* and *scp_ui* are run). A worksheet function named *include* also exists.
- *data* - specifies directory, relative to the *include* option, that contains data for the *Rhythm*® engine.
- *system* - specifies directory, relative to the *include* option, that contains data required to get *Rhythm*® running.
- *user_data* - specifies directory, relative to the *include* option, that contains data for the *Rhythm*® engine. Intended as a hook for doing demos.

4.5 Modelling Data

When the *Rhythm*® engine is run, log messages appear such as:

Reading file system/user.dat using user

This message indicates:

- Reading the *user.imp* file
- Reading the *user.dat* file which is specified by the *file* command in the *user.imp* file. Each TAB delimited (or other *delimiter* character) field in the *.dat* file is defined by the field definitions (= # ;) in the *.imp* file.
- The data is read into the model specified by the *model* command in the *.imp* file. The model command specifies in what order the *.imp* file will be loaded when loading the dataset.

4.6 Import File Layout Formats

4.6.1 Introduction

An import file can be presented with one of the following layouts for importing data.

4.6.2 import_text_file

A layout is defined as an arrangement of controls for *display* in a report. The *import_text_file* layout is different from standard layouts in that it does not deal with displaying anything, but deals with I/O (input / output). It allows the import file to import fields from a text file. The user declares the fields which are associated with an input record field. This is similar to specifying which cells go in the x axis on an axis-cross layout. # in the specified cells returns the import value for that field. If an import field is needed by multiple cells, then the cell can be referenced by name.

```
import_text_file   location()
{
    file: "site.dat";
worksheet ()
{
    model: "Location";              // Required
    sf_const Supply_Chain  SUPP = find(supply_chains, "Rhythm Demo");
    this = SUPP.sites;

    // Define fields that exist in data file:
    [ A1: name = #; ]
    [ B1: locations.name = A1; ]
    [ C1: locations.super_location = #; ]
}
import_record: A1 B1 C1 D1 E1;
}
```

4.6.3 import_dialog

The *import_dialog* import file layout sets up a report that allows editing a number of fields off-line, i.e. in the GUI only. Selecting *OK* imports all of those records.The *import_dialog* import file layout composes spreadsheet-style prompt labels of an import / export worksheet into a dialog-like layout. All the cells in the resultant layout can be edited without sending anything to the engine.

The *import_dialog* import file layout can specify a title for each cell being imported. For example:

```
[site.title   = "Site";]
[B1.title     = "Request Name";]
[C1.title     = "Item";]
[D1.title     = "Quantity";]
[E1.title     = "Due Date"; format = "MMM DD YY";]
```


[F1.title = "Description";]

- Specify the fields to be imported. A title, and a blank in which to enter a value for each field is displayed.
- Select the *Apply* button to read each of the fields from one data record (line) in a file, and set them into the models. *Apply* effectively does an export and displays the current values. Thus, if the request name is supplied, then all current values of that request are supplied. Any number of the fields can be edited, followed by selecting *OK*. This dialog does not allow changing any names. Only new names can be imported.
- Select *OK* to perform an action similar to reading a single record out of a data file with an import file that specifies those same fields.

This feature is useful for allowing an engine to have a number of order entry people connected who are each taking orders over the phone. Only one person at a time can do the ATP quote. The others are not locked out from editing the order that they are taking over the phone for that duration. They can edit freely off-line, and only wait when they select *OK*.

4.7 Import File Format

4.7.1 Import File Format

4.7.1.1 Description

The general format for the import file follows. Characters in **bold** are typed as is. Characters within <angle brackets> are user specified values:

```
import_text_file  <import_file_name> (parameters)
{
  file:  "<datafile_name1> [, ..., <datafile_namen>]"
  delimiter: "<char>";
worksheet ()
{
   <decl1>;
   <decl2>;
  [ <cell1>: <input specification1>; ]
        .
        .
        .
   [ <celln>: <input specificationn>; ]
}
import_record: <cell1 cell2 . . .celln>
}
```

where:

- *decl* - one of the following worksheet properties or variables:
  - *model*
  - *sequence*
  - *sf_const*
  - *read*
  - *this*
  - *amd*
- *input specification* - an expression formatted as follows:
  - lvalue= {# | <expression>};

4.8 Parts of the Import File Format

Table 2 describes each part of the import file format.

Table 2: Parts of the Import File Format

Part	Description
import_text_file	Layout type used for import files.
import_file_name	Name of the import file (*.imp* extension).
parameters	
; (semicolon)	indicates the end of each import file statement.
file	Name of the input data file pathname(s) relative to the import file directory. The data file name has a *.dat* extension. A layout property. The *file* property specifies which data file this import file interprets during an import operation. The import file is the data file to be read. This property can specify one or more files. *file: "<datafile_name$_1$> [, ..., <datafile_name$_n$>]"*
delimiter	Field separator that occurs in input data files. Any character may be used, including special characters. Default is the TAB character ("\t"). A layout property. *delimiter: "<char>"* Leading and trailing white space (the white space that exists between character strings and delimiters) is stripped. The following is an example of a data file which is parsed with *delimiter: ";"*. *# Comment lines begin with "#"* *# This is a data file for some supply chains* *# Name Description* *#----------* *supp_chain_1; Some supply chain* If leading and trailing whitespace was not stripped, then the description would be " Some supply chain". But the description is actually "Some supply chain".
worksheet	

Table 2: Parts of the Import File Format

Part	Description
model	Model type (from the Model Reference Manual) that is associated with the import file. It defaults to the data type of the *this* variable. A worksheet property. This property is used to sort which fields are read first, and to specify in what order the *.imp* file will be loaded when loading the dataset. For example, if the *model* is *Location*, then the locations field *name* in the site data file will be read before the site and supply chain fields (*SUPP.site.locations.name*). See the details for the *sequence* property for the sort order. This property may be any model type whose owner (or owner's owner, etc.) is the *this* model type. (The model type of *Location* has an owner of *Site*, and its owner's owner (*Site*'s owner) is *Supply_Chain*). *model: "<model_name>"* This property does not prohibit reading / writing other models. It merely tells *Rhythm*® to process this file during the time when *Rhythm*® is processing imports for the specified model. The importance of this processing is that models are always read/written in a certain order. The import file should be processed when it is time to process the model for import. The model for an import file determines: • the depth at which the data file is read • the model in which to look to get the *sequence* property
sequence	When importing data, all import files are read, and then sorted by the following: • First key: Depth in the model tree, root models (those with owner void) come first. • Second key: The *sequence* model property + the *sequence* import file property. Small numbers come before large numbers. • Third key: alphabetically by model name. *sequence: <float>* The *sequence* specified in the *.mod* files is added to the *sequence* in the import file, to give the import file writer the opportunity to override what is in the *.mod* files. Usually, import file writers will want to add a fractional to the *.mod* file's value, to resolve ambiguities between two files loading the same model. A worksheet property. An example is a file that specifies product families, and another that specifies actual products. Both use the same model, but the second references products defined in the first. To ensure the actual-product file gets read second, the user can put the following in the import file: *sequence: 0.5;*

Table 2: Parts of the Import File Format

Part	Description
sf_const	Local constant variable. The constant variables are set once at the beginning of file reading or writing. It is more efficient to use *sf_const* than *read* because it saves looking up the supply chain (which does not change) for every record. The format is as follows: *sf_const* <model-type> <name> = <expression>; For example, *sf_const Supply_Chain SUPP = find(supply_chains, "Actual");*
read	Local variable set when reading. The *read* variables are set after each record is read, and before the various cell *set* expressions are evaluated (so the *read* variable can be used by those expressions). The format is as follows: *read* <model-type> <name> = <expression>; In this example, import file variable named SUPP is set to the "Actual" Supply_Chain. *read Supply_Chain SUPP = find(supply_chains, "Actual");*
amd	Special local variable which determines what to do when reading. The available values are: • *add* - report errors if the model-instance does not exist • *add_modify* - modify existing records, and create new ones (default value) • *delete* - delete the innermost sub-model specified and ignore all other fields • *modify* - modify existing records, and report errors if they do not exist For example: *amd = add_modify;*
this	Stores the root model being read into by an import file. It is a special local variable which is the default first parameter in all read and write expressions. Its value is specified as a collection, but it is used as a member of the collection (usually one which is constructed while file reading). Required input (no default value). *this* determines the depth (model sub-level) at which data fields are being read or written by the =# read expressions. For example: *this = SUPP.sites;* *[A1: locations.name = #;]* The data field value is being read into or written from *this.name*, i.e. from *SUPP.sites.name*. The names being read would be at the site model sub-level. If *this* was defined as *this=SUPP*, then the data field value is being read into or written from *this.name*, i.e. from *SUPP.name*. In this case, the names being read would be at the supply chain model level.

Table 2: Parts of the Import File Format

Part	Description
input specification	The format for input specifications is as follows. Optional parts are in {curly brackets} # is the input record field converted to the data-type of the expression; it is non-nameable, i.e. cannot have something like #i. [<cell name:> lvalue= {# | <expression>};] For example: *[A1: sites.name =#;]* *[B1: sites.name = "Dallas";]*
import_record	Specifies the input order of fields from each record / line of an import data file. This allows the user to explicitly define the order without being constrained by the order in which the lines with #'s appear in the import file. Not required. If not specified, then no fields are read. A layout property. *import_record:* $<cell_1\ cell_2 \ldots cell_n>$ For example: *[A1: name = #;]* *[B1: buffers.item = #;]* *[C1: buffers.description = #;]* . . . **import_record***: A1 B1;* Note that more fields may be defined than are used (i.e. specified by the *import_record* property). C1 is defined but is not used in the example.

4.8.1 Import File Expressions

Table 3 describes how to specify expressions in an import file for certain tasks.

Table 3: Import File Expressions

TO specify ...	USE...
format for a field read from a data record	*[<cell name>: <field name> = #;* **format** = *"\S";*] Examples include the following: *[A1: name = #;* **format** = *"\S";]* *[E1: requests.delivery_requests.due = #;* **format** = *"MM/DD/YY";]*
import file variable as a quantity	*[<cell name>: Quantity(#);]* In the following example, C1 is a quantity: *[C1: Quantity(#);]*
import file variable as a date	*[<cell name>: Date(#, <format>);*] In the following example, C1 is a date parsed with a format named *wide:* *[D1: Date(#, wide);]*

4.8.2 Special Characters

The following special characters are used to read and write data files.

#	an input field from a data record
= #	read expression
&	concatenate
#;	skip that field
:	separates a cell name (local variable) from a read expression

4.8.3 User Defined Variables

Users may define their own variables in an import file. A cell name (e.g. A1) followed by a colon defines a local variable. In the following example, # stands for the input field (non-nameable, i.e. cannot have something like #i):

[A1: #;]

To specify an optional name that may be used for the worksheet cell, the following format can be used:

[A1 foo: #;]

is a special character used to read data files. The cell name preceding a # makes the cell name a user defined local variable. It takes the value of that field read from the data record. This value can be assigned elsewhere in the import file by specifying the cell name in an expression. See B1 in the following example:

[B1 :buffers.item = #;]
[E1: buffers.delivery= "DELIVER-" & B1;]

A sample import file using user defined local variables might appear as in the following example. Note that the second field read is *buffers.item*. Its value is read into the local variable B1, whose value is:

- assigned to *buffers.name*
- concatenated with DELIVER- and then assigned to *buffers.delivery*
- concatenated with GET- and then assigned to *buffers.supplying_operation*

```
import_text_file   buffer()
{
    file: "buffer.dat";
worksheet ()
{
    model: "Buffer";                    // Required
    sf_const Supply_Chain  SUPP = find(supply_chains, "Rhythm Demo");
    this = SUPP.sites;

// Define fields that exist in data file:
    [ A1: name = #; ]
    [ B1: buffers.item = #; ]
    [ C1: buffers.description = #; ]
    [ D1: buffers.name          = B1; ]
    [ E1: buffers.delivery      = "DELIVER-" & B1; ]
    [ F1: buffers.discrete      = "TRUE"; ]
    [ G1: buffers.flow_policy   = "BASIC"; ]
    [ H1: buffers.supplying_operation= "GET-" & B1; ]
}
import_record: A1 B1 C1;
}
```

4.8.4 Using Intermediate Values

When a cell name defined in one expression is referenced by another expression, the computed value from the expression is the one being referenced. For example:

```
[ A1: site = "From" & #; ]
[ B1: delivery_date = sellers.find(A1); ]
```

A1 gets prefixed with a text string, in this case *"From"*. If the text string prefix is not needed for the list in sellers, then the following expression should be used:

```
[ A1: "" = #;]                    // the default syntax to read a value into a cell
[ A2: site = "From" & A1; ]
[ B1: delivery_date = sellers.find(A1); ]
```


4.8.5 If

The *if* statement may be used to set the value of a field based on values in other fields. (fields can be named). See the *if* statements in the following example. The first *if* changes an extension selector depending on whether or not the *parent* field was specified. The second *if* changes the value of the *count* field depending on the value for the *location* field. Note that Chicago counts twice (#*2).

```
import_text_file   routing (Factory fm)
{
    file: "routing.dat";          // Data files to be read (may be more than 1)
worksheet ()
{
    this = fm.routings;

    [ A1:  routing = if (C1.exists, "Complex_Routing", "Simple_Routing"); ]
    [ B1:  name = #; format = "\S"; ]
    [ C1:  parent = #; ]
    [ D1:  count = if (E1 == "Chicago", #*2, #); ]
    [ E1:  location = # ? "Dallas"; ]
    [ F1:  operations.description = #; ]
    [ G1:  operations.name = #; ]   // operations is a sub-model; name is a key field
    [ H1:  effectivity.start = #; ]
    [ I1:  effectivity.time = "4 week"; ]
    [ J1:  cost = #; ]
    [ K1:  supervisor = #; ]
    [ L1:  duration = #; ]

}
import_record: A1 B1 C1 D1 E1 F1 G1 H1 I1 J1 K1 L1;
}
```

4.8.6 UI Only Fields

Most models (those with a *plist* field) can be extended with additional user defined fields. The following data file adds additional fields to various models. Each model in the following file has a name, value-type, and description. The implementation of the file stores the field data in the model's *plist*, which is essentially a vector of name/value pairs.

Model	*Value_Type*	*Field_Name*	*Description*
Data;	*Computed_String;*	*description;*	
Buffer;	*Symbol;*	*short_name;*	*Buffer category*
Item;	*Symbol;*	*short_name;*	*Buffer category*
Product;	*Symbol;*	*short_name;*	*Buffer category*
Product_Group;	*Symbol;*	*short_name;*	*Buffer category*

4.8.6.1 Read Example

The following example has two *sf_const* statements: the first specifies the supply chain name for which data will be read. The second specifies a site name that will also be a super site name:

```
import_text_file   Item_Manufacturer ()
{
    file: "mfg_item.dat";
worksheet ()
{
    model: "Item";

    sf_const Supply_Chain    SUPP= find(supply_chains, "DP Demo");
    sf_const Site            SITE= find(SUPP.sites, "Manufacturers");
}
import_record: A1 B1 C1;
}
```

Note the following details:

- Import file variable named SUPP is set to the *DP Demo* Supply_Chain by the first *sf_const* statement, i.e. SUPP has a value of *DP Demo*
- *DP Demo* has been read as the *name* of a Supply_Chain (by a *Supply_Chain.imp* import file from a *supply_chain.dat* data file)
- Import file variable named SITE is set to the *Manufacturers* Site by the second *sf_const* statement
- The sites are read into the *DP Demo* Supply_Chain (SUPP)

4.8.6.1.1 Model Property Example

Assume the (partial) *supply_chain* import file shown. The data type of the *this* variable is *supply_chains*. Since the supply chain data file is to be read into the *Supply_Chain* model (the depth), i.e. the type of model is the same as the type of *this*, the *model* property is not required:

```
import_text_file   supply_chain()
{
    file: "supply_chain.dat";
    delimiter: "\t";
worksheet ()
{
    model: "Supply_Chain";  // Optional
    this = supply_chains;
}
import_record: A1 B1 C1;
}
```

Assume the (partial) *site* import file shown. The data type of the *this* variable is *sites*. Since the site data file is to be read into the *Site* model (the depth), i.e. the type of model is the same as the type of *this*, the *model* property is not required:

```
import_text_file   site()
{
    file: "site.dat";
    delimiter: "\t";
worksheet ()
{

    model: "Site";              // Optional
    this = SUPP.sites;
}
import_record: A1 B1 C1;
}
```

Assume the (partial) *location* import file shown. The data type of the *this* variable is *sites*. Since the site data file is to be read into the *Location* model (the depth), i.e. the type of model is not the same as the type of *this*, the *model* property is required:

```
import_text_file   location()
{
    file: "site.dat";
    delimiter: "\t";
worksheet ()
{
    model: "Location";          // Required
    this = SUPP.sites;
}
import_record: A1 B1 C1;
}
```


Table 4 summarizes the previous examples.

Table 4: Model Property

owner's owner	owner	this	model
		supply_chains	Supply_Chain
	Supply_Chain	sites	Site
Supply_Chain	Site	sites	Location
Supply_Chain	Site	sites	Location (and Site)

4.8.6.1.2 Reading Two Models

To read two models from one data file, assume the (partial) *Location* import file shown. The data type of the *this* variable is *sites*. Since the site data file is to be read into the *Location* model (the depth) as well as the *Site* model, the type of model is not the same as the type of *this*, so the *model* property is required. The fields *name*, *super_location*, and *description* are read for the *Location* model. The fields *name* and *role* are read for the *Site* model:

```
import_text_file   location()
{
    file: "site.dat";
worksheet ()
{
    model: "Location";                    // Required
    this = SUPP.sites;

    // Define fields that exist in data file:
    [ A1: name = #; ]
    [ B1: locations.name = #; ]
    [ C1: locations.super_location = #; ]
    [ D1: role = #; ]
    [ E1: locations.description = #; ]
}
import_record: A1 B1 C1 D1 E1;
}
```


4.9 Sample Import Files

4.9.1 Import File Structure

The set of import file commands that are available, an explanation of each, and some representative expressions are presented in the form of a sample import file (See FIGURE 5). Import files may be derived from this structure. See succeeding sections for example import files. Properties and special local variables are shown in **bold** text.

FIGURE 5 Import File Structure

```
import_text_file   location()                  // Import file name, and one passed parameter (if any)
// Rhythm® spec file for location data file
{ delimiter: "\t";                             // Default is "\t" from the delimiter option
  file: "site.dat";                            // Specifies input data (.dat) file pathname(s), relative to the import file
                                               //    directory (may be more than 1, separated by commas)
  worksheet ()
  {
    before_import : "shell command";           // Executes a shell command before reading files
    after_import  : "shell command";           // Executes a shell command after reading files
    model: "Location";                         // Specifies model associated with a data file
                                               //    Optional; defaults to model type of this; note that here it (Location) is
                                               //    different than the model type of this (sites).
                                               //    Used only to sort which fields are read first.
                                               //    Short for this.model = Location.
    amd = add;                                 // Special local variable which determines what to do when reading
                                               //    amd (Add, Modify, Delete) defaults to add_modify
                                               //    add - report errors if the model-instance does not exist
                                               //    add_modify - modify existing records, and create new ones (default)
                                               //    delete - delete the innermost sub-model specified; ignore all other fields
                                               //    modify - modify existing records, and report errors if they do not exist

    sf_const  Supply_Chain SUPP = find(supply_chains, "Actual");
                                               // Find the Actual supply chain in the supply_chains instance of model
                                               //    Supply_Chain. Import file variable named SUPP is set to the Actual
                                               //    Supply_Chain. This is done after each record is read (so the #id
                                               //    variables are set), and before the various cell set expressions are
                                               //    evaluated (so the read variables can be used by those expressions).

    this = SUPP.sites;                         // Specifies instance of the model to be read or written

  // Define fields that exist in data file:
                                               // Fields delimited by semicolon
    [ A1: name = #; ]                          // Name for the site. Short for this.name; i.e. SUPP.site.name;
    [ B1: locations.name = #; ]                // Name for the location. Short for this.locations.name;
                                               //    i.e. SUPP.sites.locations.name;
    [ C1: locations.super_location = #; ]      // Name for the super location.
```

```
  [ D1: "" = #; ]                    // Ignore this positional input field on read; output empty-string on write

  [ E1: locations.description = # ? "Default Description"; ]
                                     // Expression is get formula for reading
                                     // # is current input (the get formula)
                                     //    the If Exists operator (?) returns the value on its right.

  [ F1: quantity = 123; ]            // Value is NOT read: it is a constant 123
  [ G1: frobitz - 22 = #; ]          // Field has 22 subtracted (get) when reading, added (set) on writing

  [ H1: sub(0, 10, order_id) = #; ]  // Concatenate 2 fields on read

  [ I1: update = #; format = ("MMDDYY hh:mm"); ]  // Reads ui-only field with the specified format

  [ J1: operations.name = #; ]       // Use this form for a single operations sub-model
  [ K1: operations[first].name = #; ]  // Use this form when reading 2 sub-models
  [ L1: operations[second].name = #; ]
}
import_record: A1 B1 C1 D1 E1 F1 G1 H1 I1 J1 K1 L1;
}
```


4.9.2 Examples

To illustrate the use of import files, and how they relate to each other and to data files, import files for *supply_chain.imp*, *site.imp*, and *location.imp* are presented. Refer to the previous section titled Sample Data Files for the *.dat* files that are mentioned. Properties and special local variables are shown in **bold** text. See FIGURE 6 through FIGURE 11.

FIGURE 6 Supply Chain Import File

```
import_text_file   supply_chain()
{
  file: "supply_chain.dat";              // Specifies input file
worksheet ()
{
  model: "Supply_Chain";                 // Specifies model associated with a data file. Optional in this case.

  this = supply_chains;                  // Specifies instance of the model to be read or written

// Define fields that exist in data file:
  [ A1: name = #; ]                      // Name for the supply chain. Short for this.name;
  [ B1: description = #; ]               // Long description for the supply chain
}
import_record: A1 B1;
}
```

Notes for *supply_chain.imp*:

1. Search all pathnames indicated by the *include* engine option for *.imp* files.
2. Read all of the *imp* files.
3. Read the *supply_chain.imp* file.
4. Read the *supply_chain.dat* file which is specified by the **file** command.
5. Each line (record) of the *.dat* file is read one at a time (into *Supply_Chain* **model**).
6. Each line (record) defines a different *Supply_Chain* **model**.
7. Each TAB delimited field in the *.dat* file is defined by, and so read as, the field definitions (=#;) in the *.imp* file.
8. The fields are keywords that exist in the model. See the *Supply Chain* top level model in the *Rhythm*® Model Reference Manual for all possible fields into which data may be read.
9. In this example, the *supply_chain.dat* file contains data that is read into the fields *name* and *description* in the *Supply Chain* model.
10. For example, the first supply chain *name* is read as *Actual*, and its *description* is read as *Our actual supply chain model.* The second *name* and *description* are read as *New Slitter* and *What-if model with additional Slitter*, respectively.

FIGURE 7 Supply Chain Data File

name	*description*
Actual	*Our actual supply chain model.*
New Slitter	*What-If model with additional Slitter.*
New DC	*What-If model with additional Distribution Center.*

FIGURE 8 Site Import File

```
import_text_file   site()
{
  file: "site.dat";                    // Specifies input file
worksheet ()
{
  model: "Site";                       // Specifies model associated with a data file. Optional in this case.
  read Supply_Chain SUPP = find(supply_chains, "Actual");
                                       // Find the Actual supply chain in the supply_chains instance of model
                                       //    Supply_Chain. Call this model SUPP.

  this = SUPP.sites;                   // Specifies instance of the model to be read or written

// Define fields that exist in data file:
  [ A1: "" = #; ]                      // See location.imp
  [ B1: name = #; ]                    // Name for the site. Short for this.name; i.e. SUPP.site.name;
  [ C1: "" = #; ]
  [ D1: role #; ]                      // Name for the role
  [ E1: description = #; ]
  [ F1: forecast_horizon = "MONTHS"; ]
}
import_record: A1 B1 C1 D1 E1 F1;
}
```

Notes for *site.imp*:

1. Read the *site.imp* file.
2. Read the *site.dat* file which is specified by the **file** command.
3. Each line (record) of the *.dat* file is read one at a time (into *Site* **model**).
4. Each line (record) defines a different *Site* **model**.
5. Each TAB delimited field in the *.dat* file is defined by, and so read as, the field definitions (=#;) in the *.imp* file.
6. The fields are keywords that exist in the model. See the *Site* sub-model of top level model *Supply Chain* in the *Rhythm*® Model Reference Manual for all possible fields into which data may be read.
7. In this example, the *site.dat* file contains data that is read into the fields *name, role,* and *description* in the *Site* model. The first and third fields are ignored.
8. For example, the first site *name* is read as *Actual*, its role is read as LINK, and its *description* is read as *Super Site for all.* The second *name, role,* and *description* are read as *COSD, LINK,* and *Commercial Office Supply*, respectively.
9. A global default value of "MONTHS" is supplied for the field *forecast_horizon* for all site records read into the *Site* model.

FIGURE 9

Site Data File

site	*name(site/loc)*	*super_location*	*role*	*description*
Actual	*Actual*	*[unspecified]*	*LINK*	*Super Site for all*
#				
Actual	*COSD*	*Actual*	*LINK*	*Commercial Office Supply*
Actual	*CSTD*	*Actual*	*LINK*	*Consumer Stationary Tape*
Actual	*ISTD*	*Actual*	*LINK*	*Industrial Stationary Tape*
#				
Actual	*Atlanta DC*	*Actual*	*LINK*	*serves Southeast Region*
Actual	*Dallas DC*	*Actual*	*LINK*	*serves Southwest Region*
Actual	*Ontario DC*	*Actual*	*LINK*	*serves Pacific Region*
Actual	*Dekald DC*	*Actual*	*LINK*	*serves Northern Region*
Actual	*New Jersey DC*	*Actual*	*LINK*	*serves Northeast Region*
#				
Actual	*Hutchison AT*	*Actual*	*LINK*	*Adhesive Tape Plant*
Actual	*Hutchison MT*	*Actual*	*LINK*	*Magnetic Tape Plant*
Actual	*Cynthia*	*Actual*	*LINK*	*Film/Jumbo Plant*
Actual	*Cottage Grove*	*Actual*	*LINK*	*Glue/Adhesive Plant*
Actual	*Chicago*	*Actual*	*LINK*	*Post-it Note Plant*
#				
Actual	*Pure Chemicals*	*Actual*	*SUPPLIER*	*Chemical Supplier*
Actual	*Global Plastics*	*Actual*	*SUPPLIER*	*Plastics Supplier*

FIGURE 10 Location Import File

```
import_text_file   location()
{
  file: "site.dat";                          // Specifies input file
worksheet ()
{
  model: "Location";                         // Specifies model associated with a data file. Required in this case.
  read Supply_Chain SUPP = find(supply_chains, "Actual");
                                             // Find the Actual supply chain in the supply_chains instance of model
                                             //    Supply_Chain. Call this model SUPP.

  this = SUPP.sites;                         // Specifies instance of the model to be read or written

// Define fields that exist in data file:
  [ A1: name = #; ]                          // Name for the site. Short for this.name; i.e. SUPP.site.name;
  [ B1: locations.name = #; ]                // Name for the location. Short for this.locations.name;
  [ C1: locations.super_location = #; ]      // Name for the super location.
  [ D1: "" = #; ]                            // Ignore this positional input field on read; output empty-string on write
  [ E1: locations.description = #; ]
}
import_record: A1 B1 C1 D1 E1;
}
```

Notes for *location.imp*:

1. Read the *location.imp* file.
2. Read the *site.dat* file which is specified by the **file** command.
3. Note that this is the same data file read by the *site.imp* import file. This is because the site and location share common data.
4. Each line (record) of the .dat file is read one at a time (into *Location* model).
5. Each TAB delimited field in the .dat file is defined by, and so read as, the field definitions (=#;) in the *.imp* file.
6. The fields are keywords that exist in the model. See the *Location* sub-model of model *Site* in the *Rhythm*® Model Reference Manual for all possible fields into which data may be read.
7. In this example, the *site.dat* file contains data that is read into the fields site *name,* locations *name,* locations *super_location,* and locations *description* in the *Site* model.
8. For example, the first site *name* is read as *Actual,* locations *name* is read as *Actual,* locations *super_location* is read as *[unspecified],* and locations *description* is read as *Super Site for all.* Note that the fourth field is ignored.

FIGURE 11 Site Data File

site	*name(site/loc)*	*super_location*	*role*	*description*
Actual	*Actual*	*[unspecified]*	*LINK*	*Super Site for all*
#				
Actual	*COSD*	*Actual*	*LINK*	*Commercial Office Supply*
Actual	*CSTD*	*Actual*	*LINK*	*Consumer Stationary Tape*
Actual	*ISTD*	*Actual*	*LINK*	*Industrial Stationary Tape*
#				
Actual	*Atlanta DC*	*Actual*	*LINK*	*serves Southeast Region*
Actual	*Dallas DC*	*Actual*	*LINK*	*serves Southwest Region*
Actual	*Ontario DC*	*Actual*	*LINK*	*serves Pacific Region*
Actual	*Dekald DC*	*Actual*	*LINK*	*serves Northern Region*
Actual	*New Jersey DC*	*Actual*	*LINK*	*serves Northeast Region*
#				
Actual	*Hutchison AT*	*Actual*	*LINK*	*Adhesive Tape Plant*
Actual	*Hutchison MT*	*Actual*	*LINK*	*Magnetic Tape Plant*
Actual	*Cynthia*	*Actual*	*LINK*	*Film/Jumbo Plant*
Actual	*Cottage Grove*	*Actual*	*LINK*	*Glue/Adhesive Plant*
Actual	*Chicago*	*Actual*	*LINK*	*Post-it Note Plant*
#				
Actual	*Pure Chemicals*	*Actual*	*SUPPLIER*	*Chemical Supplier*
Actual	*Global Plastics*	*Actual*	*SUPPLIER*	*Plastics Supplier*

4.9.3 Read Two Models

This example illustrates reading two models from one data file with one import file. See FIGURE 12.

FIGURE 12 Multiple Models

```
import_text_file   site()
{
  file: "site.dat";                          // Specifies input file
worksheet ()
{
  model: "Site";                             // Specifies model associated with a data file.
  read Supply_Chain SUPP = find(supply_chains, "Actual");
                                             // Find the Actual supply chain in the supply_chains instance of model
                                             //    Supply_Chain. Call this model SUPP.

  this = SUPP.sites;                         // Specifies instance of the model to be read or written

// Define fields that exist in data file:
  [ A1: name = #; ]                          // Name for the site. Short for this.name; i.e. SUPP.site.name;
  [ B1: locations.name = #; ]                // Name for the location. Short for this.locations.name;
  [ C1: locations.super_location = #; ]      // Name for the super location.
  [ D1: role = #; ]                          // Name for the role
  [ E1: locations.description = #; ]
}
import_record: A1 B1 C1 D1 E1;
}
```

Notes for *site.imp*:

1. Read the *site.imp* file.
2. Read the *site.dat* file which is specified by the **file** command.
3. Note that this is the same data file read by the *site.imp* import file. This is because the site and location share common data.
4. Each line (record) of the *.dat* file is read one at a time (into *Location* model).
5. Each TAB delimited field in the *.dat* file is defined by, and so read as, the field definitions (=#;) in the *.imp* file.
6. The fields are keywords that exist in the model. See the *Location* sub-model of model *Site* in the *Rhythm*® Model Reference Manual for all possible fields into which data may be read.
7. In this example, the *site.dat* file contains data that is read into the *name* and *role* fields of the *site* model, and into the *name*, *super_location*, and *description* fields in the *locations* model.

8. For example, the first site *name* is read as *Actual*, locations *name* is read as *Actual*, locations *super_location* is read as *[unspecified]*, site *role* is read as *LINK*, and locations *description* is read as *Super Site for all.*

FIGURE 13 Site Data File

site	*name(site/loc)*	*super_location*	*role*	*description*
Actual	*Actual*	*[unspecified]*	*LINK*	*Super Site for all*
#				
Actual	*COSD*	*Actual*	*LINK*	*Commercial Office Supply*
Actual	*CSTD*	*Actual*	*LINK*	*Consumer Stationary Tape*
Actual	*ISTD*	*Actual*	*LINK*	*Industrial Stationary Tape*
#				
Actual	*Atlanta DC*	*Actual*	*LINK*	*serves Southeast Region*
Actual	*Dallas DC*	*Actual*	*LINK*	*serves Southwest Region*
Actual	*Ontario DC*	*Actual*	*LINK*	*serves Pacific Region*
Actual	*Dekald DC*	*Actual*	*LINK*	*serves Northern Region*
Actual	*New Jersey DC*	*Actual*	*LINK*	*serves Northeast Region*
#				
Actual	*Hutchison AT*	*Actual*	*LINK*	*Adhesive Tape Plant*
Actual	*Hutchison MT*	*Actual*	*LINK*	*Magnetic Tape Plant*
Actual	*Cynthia*	*Actual*	*LINK*	*Film/Jumbo Plant*
Actual	*Cottage Grove*	*Actual*	*LINK*	*Glue/Adhesive Plant*
Actual	*Chicago*	*Actual*	*LINK*	*Post-it Note Plant*
#				
Actual	*Pure Chemicals*	*Actual*	*SUPPLIER*	*Chemical Supplier*
Actual	*Global Plastics*	*Actual*	*SUPPLIER*	*Plastics Supplier*

4.10 Planning with Import Files

The import files in this section provide examples of how to read data files for planning. A sample data file follows each import file. For each field in all data files, refer to the *Rhythm*® Model Reference Manual for details. For each import file, refer to the previous sample import files for explanations of various syntax rules that are implemented.

4.10.1 Distributor Buffer

This example illustrates an import file for reading a distributor buffer data file. See FIGURE 14.

FIGURE 14 Distributor Buffer Import File

```
import_text_file   Buffer_Distributor ()
{
  file: "dist_buffer.dat";
worksheet ()
{
  model: "Buffer";

  read Supply_Chain   SUPP = find(supply_chains, "DP Demo");

  this = SUPP.sites;

  [ A1: name = #; ]                                    // Site where buffer resides
  [ B1: buffers.item = #; ]                            // Item that buffer holds
  [ C1: buffers.description = #; ]                     // Long description for buffer

  [ D1: buffers.name                = B1; ]            // Short name for buffer
  [ E1: buffers.delivery            = "DELIVER-" & B1; ]  // Operation to deliver item from buffer
  [ F1: buffers.flow_policy         = "BASIC"; ]       // Distributor flow policy is BASIC
  [ G1: buffers.location            = A1; ]            // Location name is same as site
  [ H1: buffers.supplying_operation= "GET-" & B1; ]    // Operation to fill buffer
}
import_record: A1 B1 C1 D1 E1 F1 G1 H1;
}
```


FIGURE 15 Distributor Buffer Data File

site	*item*	*description*
#Houston, TX Warehouse		
HOU	*C8003001*	*Buffer for item C8003001 at Houston, TX Warehouse*
HOU	*C8003002*	*Buffer for item C8003002 at Houston, TX Warehouse*
#Los Angeles, CA Warehouse		
LOS	*C8003001*	*Buffer for item C8003001 at Los Angeles, CA Warehouse*
LOS	*C8003002*	*Buffer for item C8003002 at Los Angeles, CA Warehouse*
#London, England Warehouse		
LON	*C8003001*	*Buffer for item C8003001 at London, England Warehouse*
LON	*C8003002*	*Buffer for item C8003002 at London, England Warehouse*
#New York, NY Warehouse		
NEW	*C8003001*	*Buffer for item C8003001 at New York, NY Warehouse*
NEW	*C8003002*	*Buffer for item C8003002 at New York, NY Warehouse*
#Eastern US Market Area		
EAS	*C8003001*	*Buffer for item C8003001 at Eastern US Market Area*
EAS	*C8003002*	*Buffer for item C8003002 at Eastern US Market Area*
#European Market Area		
EUR	*C8003001*	*Buffer for item C8003001 at European Market Area*
EUR	*C8003002*	*Buffer for item C8003002 at European Market Area*
#Japanese Market Area		
JAP	*C8003001*	*Buffer for item C8003001 at Japanese Market Area*
JAP	*C8003002*	*Buffer for item C8003002 at Japanese Market Area*
#Midwestern US Market Area		
MID	*C8003001*	*Buffer for item C8003001 at Midwestern US Market Area*
MID	*C8003002*	*Buffer for item C8003002 at Midwestern US Market Area*
#Northern US Market Area		
NOR	*C8003001*	*Buffer for item C8003001 at Northern US Market Area*
NOR	*C8003002*	*Buffer for item C8003002 at Northern US Market Area*
#Southern US Market Area		
SOU	*C8003001*	*Buffer for item C8003001 at Southern US Market Area*
SOU	*C8003002*	*Buffer for item C8003002 at Southern US Market Area*
#Southwestern US Market Area		
SWS	*C8003001*	*Buffer for item C8003001 at Southwestern US Market Area*
SWS	*C8003002*	*Buffer for item C8003002 at Southwestern US Market Area*
#Western US Market Area		
WST	*C8003001*	*Buffer for item C8003001 at Western US Market Area*
WST	*C8003002*	*Buffer for item C8003002 at Western US Market Area*
#California Market Area		
CA	*C8003001*	*Buffer for item C8003001 at California Market Area*
CA	*C8003002*	*Buffer for item C8003002 at California Market Area*
#Ohio Market Area		
OH	*C8003001*	*Buffer for item C8003001 at Ohio Market Area*
OH	*C8003002*	*Buffer for item C8003002 at Ohio Market Area*
#Texas Market Area		
TX	*C8003001*	*Buffer for item C8003001 at Texas Market Area*
TX	*C8003002*	*Buffer for item C8003002 at Texas Market Area*

4.10.2 Manufacturer Buffer

This example illustrates an import file for reading a manufacturer buffer data file. See FIGURE 16.

FIGURE 16 Manufacturer Buffer Import File

```
import_text_file   Buffer_Manufacturer ()
{
  file: "mfg_buffer.dat";
worksheet ()
{
  model: "Buffer";

  read Supply_Chain   SUPP = find(supply_chains, "DP Demo");

  this = SUPP.sites;

  [ A1: name = #; ]                                  // Site where buffer resides
  [ B1: buffers.item = #; ]                          // Item that buffer holds
  [ C1: "" = #; ]
  [ D1: buffers.flow_policy = #; ]                   // Buffer flow policy
  [ E1: buffers.description = #; ]                   // Long description for buffer

  [ F1: buffers.name              = B1; ]            // Short name for buffer
  [ G1: buffers.delivery          = "DELIVER-" & B1; ]  // Operation to deliver item from buffer
  [ G1: buffers.location          = A1; ]            // Location name is same as site
  [ H1: buffers.supplying_operation= "MAKE-" & B1; ] // Operation to fill buffer
}
import_record: A1 B1 C1 D1 E1 F1 G1 H1;
}
```

Notes:

1. buffers.delivery does not appear in the assembly or raw materials buffers
2. Raw material buffer has BUY instead of MAKE

FIGURE 17 Manufacturer Buffer Data File

site	*item*	*discrete*	*flow_policy*	*description*
# Boston, MA Plant				
BOS	*C8003001*	*TRUE*	*LFL_SIMPLE*	*Buffer for C8003001 at Boston, MA Plant*
BOS	*C8003002*	*TRUE*	*LFL_SIMPLE*	*Buffer for C8003002 at Boston, MA Plant*
BOS	*C8003011*	*TRUE*	*LFL_SIMPLE*	*Buffer for C8003011 at Boston, MA Plant*
BOS	*C8003012*	*TRUE*	*LFL_SIMPLE*	*Buffer for C8003012 at Boston, MA Plant*
BOS	*T8003001*	*TRUE*	*LFL_SIMPLE*	*Buffer for T8003001 at Boston, MA Plant*
BOS	*T8003002*	*TRUE*	*LFL_SIMPLE*	*Buffer for T8003002 at Boston, MA Plant*
BOS	*T8003003*	*TRUE*	*LFL_SIMPLE*	*Buffer for T8003003 at Boston, MA Plant*
BOS	*T8003011*	*TRUE*	*LFL_SIMPLE*	*Buffer for T8003011 at Boston, MA Plant*
BOS	*T8003012*	*TRUE*	*LFL_SIMPLE*	*Buffer for T8003012 at Boston, MA Plant*
BOS	*T8003013*	*TRUE*	*LFL_SIMPLE*	*Buffer for T8003013 at Boston, MA Plant*
# Dallas, TX Plant				
DAL	*C8003001*	*TRUE*	*LFL_SIMPLE*	*Buffer for C8003001 at Dallas, TX Plant*
DAL	*C8003002*	*TRUE*	*LFL_SIMPLE*	*Buffer for C8003002 at Dallas, TX Plant*
DAL	*C8003012*	*TRUE*	*LFL_SIMPLE*	*Buffer for C8003012 at Dallas, TX Plant*
DAL	*T8003001*	*TRUE*	*LFL_SIMPLE*	*Buffer for T8003001 at Dallas, TX Plant*
DAL	*T8003002*	*TRUE*	*LFL_SIMPLE*	*Buffer for T8003002 at Dallas, TX Plant*
DAL	*T8003003*	*TRUE*	*LFL_SIMPLE*	*Buffer for T8003003 at Dallas, TX Plant*
DAL	*T8003011*	*TRUE*	*LFL_SIMPLE*	*Buffer for T8003011 at Dallas, TX Plant*
DAL	*T8003012*	*TRUE*	*LFL_SIMPLE*	*Buffer for T8003012 at Dallas, TX Plant*
DAL	*T8003013*	*TRUE*	*LFL_SIMPLE*	*Buffer for T8003013 at Dallas, TX Plant*
# Riverside, CA Plant				
RIV	*C8003001*	*TRUE*	*LFL_SIMPLE*	*Buffer for C8003001 at Riverside, CA Plant*
RIV	*C8003002*	*TRUE*	*LFL_SIMPLE*	*Buffer for C8003002 at Riverside, CA Plant*
RIV	*C8003011*	*TRUE*	*LFL_SIMPLE*	*Buffer for C8003011 at Riverside, CA Plant*
RIV	*C8003012*	*TRUE*	*LFL_SIMPLE*	*Buffer for C8003012 at Riverside, CA Plant*
RIV	*T8003001*	*TRUE*	*LFL_SIMPLE*	*Buffer for T8003001 at Riverside, CA Plant*
RIV	*T8003002*	*TRUE*	*LFL_SIMPLE*	*Buffer for T8003002 at Riverside, CA Plant*
RIV	*T8003003*	*TRUE*	*LFL_SIMPLE*	*Buffer for T8003003 at Riverside, CA Plant*
RIV	*T8003011*	*TRUE*	*LFL_SIMPLE*	*Buffer for T8003011 at Riverside, CA Plant*
RIV	*T8003012*	*TRUE*	*LFL_SIMPLE*	*Buffer for T8003012 at Riverside, CA Plant*
RIV	*T8003013*	*TRUE*	*LFL_SIMPLE*	*Buffer for T8003013 at Riverside, CA Plant*

4.10.3 Manufacturer Assembly Buffer

This example illustrates an import file for reading a manufacturer assembly buffer data file. See FIGURE 16.

FIGURE 18 Manufacturer Assembly Buffer Import File

```
import_text_file   Buffer_Manufacturer_Asm ()
{
  file: "mfg_buffer_asm.dat";
worksheet ()
{
  model: "Buffer";

  read Supply_Chain   SUPP = find(supply_chains, "DP Demo");

  this = SUPP.sites;

  [ A1: name = #; ]                                  // Site where buffer resides
  [ B1 = buffers.item; ]                             // Item that buffer holds
  [ C1: "" = #; ]                                    // Does the buffer hold whole units
  [ D1: buffers.flow_policy = #; ]                   // Buffer flow policy
  [ E1: buffers.description = #; ]                   // Long description for buffer

  [ F1: buffers.name                = B1; ]          // Short name for buffer
  [ G1: buffers.location            = A1; ]          // Location name is same as site
  [ H1: buffers.supplying_operation= "MAKE-" & B1; ] // Operation to fill buffer
}
import_record: A1 B1 C1 D1 E1 F1 G1 H1;
}
```

Notes:

1. Raw material buffer has BUY instead of MAKE

FIGURE 19 Manufacturer Assembly Buffer Data File

site	*item*	*discrete*	*flow_policy*	*description*
# Boston, MA Plant				
BOS	*ASM01000*	*TRUE*	*LFL_SIMPLE*	*Buffer for ASM01000 at Boston, MA Plant*
BOS	*ASM01200*	*TRUE*	*LFL_SIMPLE*	*Buffer for ASM01200 at Boston, MA Plant*
BOS	*ASM01300*	*TRUE*	*LFL_SIMPLE*	*Buffer for ASM01300 at Boston, MA Plant*
BOS	*MFT01100*	*TRUE*	*LFL_SIMPLE*	*Buffer for MFT01100 at Boston, MA Plant*
BOS	*MFT01400*	*TRUE*	*LFL_SIMPLE*	*Buffer for MFT01400 at Boston, MA Plant*
BOS	*MFT01500*	*TRUE*	*LFL_SIMPLE*	*Buffer for MFT01500 at Boston, MA Plant*
BOS	*MFT01600*	*TRUE*	*LFL_SIMPLE*	*Buffer for MFT01600 at Boston, MA Plant*
BOS	*MFT01700*	*TRUE*	*LFL_SIMPLE*	*Buffer for MFT01700 at Boston, MA Plant*
BOS	*MFT01800*	*TRUE*	*LFL_SIMPLE*	*Buffer for MFT01800 at Boston, MA Plant*
# Dallas, TX Plant				
DAL	*ASM01000*	*TRUE*	*LFL_SIMPLE*	*Buffer for ASM01000 at Dallas, TX Plant*
DAL	*ASM01200*	*TRUE*	*LFL_SIMPLE*	*Buffer for ASM01200 at Dallas, TX Plant*
DAL	*ASM01300*	*TRUE*	*LFL_SIMPLE*	*Buffer for ASM01300 at Dallas, TX Plant*
DAL	*MFT01100*	*TRUE*	*LFL_SIMPLE*	*Buffer for MFT01100 at Dallas, TX Plant*
DAL	*MFT01400*	*TRUE*	*LFL_SIMPLE*	*Buffer for MFT01400 at Dallas, TX Plant*
DAL	*MFT01500*	*TRUE*	*LFL_SIMPLE*	*Buffer for MFT01500 at Dallas, TX Plant*
DAL	*MFT01600*	*TRUE*	*LFL_SIMPLE*	*Buffer for MFT01600 at Dallas, TX Plant*
DAL	*MFT01700*	*TRUE*	*LFL_SIMPLE*	*Buffer for MFT01700 at Dallas, TX Plant*
DAL	*MFT01800*	*TRUE*	*LFL_SIMPLE*	*Buffer for MFT01800 at Dallas, TX Plant*
# Riverside, CA Plant				
RIV	*ASM01000*	*TRUE*	*LFL_SIMPLE*	*Buffer for ASM01000 at Riverside, CA Plant*
RIV	*ASM01200*	*TRUE*	*LFL_SIMPLE*	*Buffer for ASM01200 at Riverside, CA Plant*
RIV	*ASM01300*	*TRUE*	*LFL_SIMPLE*	*Buffer for ASM01300 at Riverside, CA Plant*
RIV	*MFT01100*	*TRUE*	*LFL_SIMPLE*	*Buffer for MFT01100 at Riverside, CA Plant*
RIV	*MFT01400*	*TRUE*	*LFL_SIMPLE*	*Buffer for MFT01400 at Riverside, CA Plant*
RIV	*MFT01500*	*TRUE*	*LFL_SIMPLE*	*Buffer for MFT01500 at Riverside, CA Plant*
RIV	*MFT01600*	*TRUE*	*LFL_SIMPLE*	*Buffer for MFT01600 at Riverside, CA Plant*
RIV	*MFT01700*	*TRUE*	*LFL_SIMPLE*	*Buffer for MFT01700 at Riverside, CA Plant*
RIV	*MFT01800*	*TRUE*	*LFL_SIMPLE*	*Buffer for MFT01800 at Riverside, CA Plant*

4.10.4 Manufacturer Raw Material Buffer

This example illustrates an import file for reading a manufacturer raw material buffer data file. See FIGURE 16.

FIGURE 20 Manufacturer Raw Material Buffer Import File

```
import_text_file   Buffer_Manufacturer_Raw ()
{
  file: "mfg_buffer_raw.dat";
worksheet ()
{
  model: "Buffer";

  read Supply_Chain   SUPP = find(supply_chains, "DP Demo");

  this = SUPP.sites;

  [ A1: name = #; ]                                  // Site where buffer resides
  [ B1: buffers.item = #; ]                          // Item that buffer holds
  [ C1: "" = #; ]                                    // Does the buffer hold whole units
  [ D1: buffers.flow_policy = #; ]                   // Buffer flow policy
  [ E1: buffers.description = #; ]                   // Long description for buffer

  [ F1: buffers.name              = B1; ]            // Short name for buffer
  [ G1: buffers.location          = A1; ]            // Location name is same as site
  [ H1: buffers.supplying_operation= "BUY-" & B1; ]  // Operation to fill buffer
}
import_record: A1 B1 C1 D1 E1 F1 G1 H1;
}
```

Notes:

1. Raw material buffer has BUY instead of MAKE

FIGURE 21 Manufacturer Raw Material Buffer Data File

site	*item*	*disc*	*flow_policy*	*description*
# Boston, MA Plant				
BOS	*MAT20000*	*FALSE*	*BASIC*	*Buffer for MAT20000 at Boston, MA Plant*
BOS	*MAT20100*	*FALSE*	*BASIC*	*Buffer for MAT20100 at Boston, MA Plant*
BOS	*MAT30010*	*FALSE*	*BASIC*	*Buffer for MAT30010 at Boston, MA Plant*
BOS	*MAT30020*	*FALSE*	*BASIC*	*Buffer for MAT30020 at Boston, MA Plant*
BOS	*MAT30021*	*FALSE*	*BASIC*	*Buffer for MAT30021 at Boston, MA Plant*
BOS	*MAT40010*	*TRUE*	*BASIC*	*Buffer for MAT40010 at Boston, MA Plant*
BOS	*MAT40020*	*TRUE*	*BASIC*	*Buffer for MAT40020 at Boston, MA Plant*
BOS	*MAT40110*	*TRUE*	*BASIC*	*Buffer for MAT40110 at Boston, MA Plant*
BOS	*MAT40120*	*TRUE*	*BASIC*	*Buffer for MAT40120 at Boston, MA Plant*
BOS	*MAT40041*	*TRUE*	*BASIC*	*Buffer for MAT40041 at Boston, MA Plant*
BOS	*MAT40042*	*TRUE*	*BASIC*	*Buffer for MAT40042 at Boston, MA Plant*
# Dallas, TX Plant				
DAL	*MAT20000*	*FALSE*	*BASIC*	*Buffer for MAT20000 at Dallas, TX Plant*
DAL	*MAT20100*	*FALSE*	*BASIC*	*Buffer for MAT20100 at Dallas, TX Plant*
DAL	*MAT30010*	*FALSE*	*BASIC*	*Buffer for MAT30010 at Dallas, TX Plant*
DAL	*MAT30020*	*FALSE*	*BASIC*	*Buffer for MAT30020 at Dallas, TX Plant*
DAL	*MAT30021*	*FALSE*	*BASIC*	*Buffer for MAT30021 at Dallas, TX Plant*
DAL	*MAT40010*	*TRUE*	*BASIC*	*Buffer for MAT40010 at Dallas, TX Plant*
DAL	*MAT40020*	*TRUE*	*BASIC*	*Buffer for MAT40020 at Dallas, TX Plant*
DAL	*MAT40110*	*TRUE*	*BASIC*	*Buffer for MAT40110 at Dallas, TX Plant*
DAL	*MAT40120*	*TRUE*	*BASIC*	*Buffer for MAT40120 at Dallas, TX Plant*
DAL	*MAT40041*	*TRUE*	*BASIC*	*Buffer for MAT40041 at Dallas, TX Plant*
DAL	*MAT40042*	*TRUE*	*BASIC*	*Buffer for MAT40042 at Dallas, TX Plant*
# Riverside, CA Plant				
RIV	*MAT20000*	*FALSE*	*BASIC*	*Buffer for MAT20000 at Riverside, CA Plant*
RIV	*MAT20100*	*FALSE*	*BASIC*	*Buffer for MAT20100 at Riverside, CA Plant*
RIV	*MAT30010*	*FALSE*	*BASIC*	*Buffer for MAT30010 at Riverside, CA Plant*
RIV	*MAT30020*	*FALSE*	*BASIC*	*Buffer for MAT30020 at Riverside, CA Plant*
RIV	*MAT30021*	*FALSE*	*BASIC*	*Buffer for MAT30021 at Riverside, CA Plant*
RIV	*MAT40010*	*TRUE*	*BASIC*	*Buffer for MAT40010 at Riverside, CA Plant*
RIV	*MAT40020*	*TRUE*	*BASIC*	*Buffer for MAT40020 at Riverside, CA Plant*
RIV	*MAT40110*	*TRUE*	*BASIC*	*Buffer for MAT40110 at Riverside, CA Plant*
RIV	*MAT40120*	*TRUE*	*BASIC*	*Buffer for MAT40120 at Riverside, CA Plant*
RIV	*MAT40041*	*TRUE*	*BASIC*	*Buffer for MAT40041 at Riverside, CA Plant*
RIV	*MAT40042*	*TRUE*	*BASIC*	*Buffer for MAT40042 at Riverside, CA Plant*

4.10.5 Forecast

This example illustrates an import file for reading a forecast data file. See FIGURE 22.

FIGURE 22 Forecast Import File

```
import_text_file   Forecast ()
{
  file: "forecast.dat";
worksheet ()
{
  model: "Forecast";

  read Supply_Chain   SUPP = find(supply_chains, "DP Demo");

  this = SUPP.plans;

  [ A1: name = #; ]                                          // Short name for the plan
  [ B1: seller_plans.seller = #; ]                           // Seller the seller plan is for
  [ C1: seller_plans.forecasts.name = #; ]                   // Product group forecast if for
  [ D1: seller_plans.forecasts.entries.period = #; ]         // Period the forecast entry is for
  [ E1: seller_plans.forecasts.entries.forecasted = #; ]     // Quantity forecasted during period

  [ F1: seller_plans.forecasts.use_std_split = "TRUE"; ]     // Always use standard split
  [ G1: seller_plans.forecasts.entries.committed = E1 * 0.9; ] // Standard commitment is 90% of forecast
}
import_record: A1 B1 C1 D1 E1 F1 G1;
}
```


FIGURE 23 Forecast Data File

plan	*seller*	*product*	*forecast-period*	*value*
Active	*CEN*	*CHA*	*07/01/95 00:00:00 / 08/01/95 00:00:00*	*100*
Active	*CEN*	*CHA*	*08/01/95 00:00:00 / 09/01/95 00:00:00*	*100*
Active	*EAS*	*CHA*	*07/01/95 00:00:00 / 08/01/95 00:00:00*	*110*
Active	*EAS*	*CHA*	*08/01/95 00:00:00 / 09/01/95 00:00:00*	*110*
Active	*EUR*	*CHA*	*07/01/95 00:00:00 / 08/01/95 00:00:00*	*200*
Active	*EUR*	*TAB*	*10/01/95 00:00:00 / 11/01/95 00:00:00*	*150*
Active	*JAP*	*TAB*	*09/01/95 00:00:00 / 10/01/95 00:00:00*	*80*
Active	*JAP*	*TAB*	*10/01/95 00:00:00 / 11/01/95 00:00:00*	*80*
Active	*MID*	*TAB*	*09/01/95 00:00:00 / 10/01/95 00:00:00*	*120*
Active	*MID*	*TAB*	*10/01/95 00:00:00 / 11/01/95 00:00:00*	*120*
Active	*NOR*	*TAB*	*09/01/95 00:00:00 / 10/01/95 00:00:00*	*150*
Active	*SOU*	*CHA*	*12/01/95 00:00:00 / 01/01/96 00:00:00*	*130*
Active	*SOU*	*TAB*	*07/01/95 00:00:00 / 08/01/95 00:00:00*	*110*
Active	*SWS*	*CHA*	*12/01/95 00:00:00 / 01/01/96 00:00:00*	*50*
Active	*SWS*	*TAB*	*07/01/95 00:00:00 / 08/01/95 00:00:00*	*80*
Active	*WST*	*CHA*	*10/01/95 00:00:00 / 11/01/95 00:00:00*	*90*
Active	*WST*	*CHA*	*11/01/95 00:00:00 / 12/01/95 00:00:00*	*90*
Active	*WST*	*CHA*	*12/01/95 00:00:00 / 01/01/96 00:00:00*	*90*
Active	*CA*	*CHA*	*09/01/95 00:00:00 / 10/01/95 00:00:00*	*50*
Active	*CA*	*CHA*	*10/01/95 00:00:00 / 11/01/95 00:00:00*	*50*
Active	*CA*	*CHA*	*11/01/95 00:00:00 / 12/01/95 00:00:00*	*50*
Active	*NY*	*CHA*	*08/01/95 00:00:00 / 09/01/95 00:00:00*	*100*
Active	*NY*	*CHA*	*09/01/95 00:00:00 / 10/01/95 00:00:00*	*100*
Active	*NY*	*CHA*	*10/01/95 00:00:00 / 11/01/95 00:00:00*	*100*
Active	*OH*	*CHA*	*07/01/95 00:00:00 / 08/01/95 00:00:00*	*60*
Active	*OH*	*CHA*	*08/01/95 00:00:00 / 09/01/95 00:00:00*	*60*
Active	*OH*	*CHA*	*09/01/95 00:00:00 / 10/01/95 00:00:00*	*60*
Active	*OH*	*TAB*	*12/01/95 00:00:00 / 01/01/96 00:00:00*	*70*
Active	*TX*	*CHA*	*07/01/95 00:00:00 / 08/01/95 00:00:00*	*180*
Active	*TX*	*CHA*	*08/01/95 00:00:00 / 09/01/95 00:00:00*	*180*
Active	*TX*	*TAB*	*11/01/95 00:00:00 / 12/01/95 00:00:00*	*170*
Active	*TX*	*TAB*	*12/01/95 00:00:00 / 01/01/96 00:00:00*	*170*
New WH	*CEN*	*CHA*	*07/01/95 00:00:00 / 08/01/95 00:00:00*	*150*
New WH	*MID*	*TAB*	*07/01/95 00:00:00 / 08/01/95 00:00:00*	*150*
New WH	*NOR*	*CHA*	*07/01/95 00:00:00 / 08/01/95 00:00:00*	*150*
New WH	*NOR*	*TAB*	*07/01/95 00:00:00 / 08/01/95 00:00:00*	*150*
New WH	*NY*	*CHA*	*07/01/95 00:00:00 / 08/01/95 00:00:00*	*150*
New WH	*NY*	*TAB*	*07/01/95 00:00:00 / 08/01/95 00:00:00*	*150*
New WH	*OH*	*CHA*	*07/01/95 00:00:00 / 08/01/95 00:00:00*	*150*
FY96	*EUR*	*TAB*	*10/01/95 00:00:00 / 11/01/95 00:00:00*	*150*
FY96	*JAP*	*CHA*	*10/01/95 00:00:00 / 11/01/95 00:00:00*	*150*
FY96	*JAP*	*TAB*	*10/01/95 00:00:00 / 11/01/95 00:00:00*	*150*
FY96	*WST*	*CHA*	*10/01/95 00:00:00 / 11/01/95 00:00:00*	*150*

4.10.6 Distributor Item

This example illustrates an import file for reading a distributor item data file. See FIGURE 24.

FIGURE 24 Distributor Item Import File

```
import_text_file   Item_Distributor ()
{
  file: "dist_item.dat";
worksheet ()
{
  model: "Item";

  read Supply_Chain   SUPP = find(supply_chains, "DP Demo");
  read Site           SITE = find(SUPP.sites, "Distributors");

// All distributor sub-sites share these items by putting them in the Distributors super-site

  this = SITE.items;

  [ A1: name = #; ]                          // Part number for item
  [ B1: "" = #;                              // Is item only in whole units
  [ C1: description = #; ]                   // Long description for item

  [ D1: artificial = "FALSE"; ]              // Distributor items are real items
}
import_record: A1 B1 C1 D1;
}
```

Notes:

1. All distributor sub-sites share these items by putting them in the Distributors super-site.

FIGURE 25 Distributor Item Data File

(End items that are requested, and promised)

name	*discrete*	*description*
C8003001	*TRUE*	*Folding Chair, Steel, Beige*
C8003002	*TRUE*	*Folding Chair, Steel, Brown*
C8003011	*TRUE*	*Folding Chair, Padded, Beige*
C8003012	*TRUE*	*Folding Chair, Padded, Brown*
T8003001	*TRUE*	*Folding Table, Laminate, 30x60*
T8003002	*TRUE*	*Folding Table, Laminate, 30x72*
T8003003	*TRUE*	*Folding Table, Laminate, 30x96*
T8003011	*TRUE*	*Folding Table, Melamine, 30x60*
T8003012	*TRUE*	*Folding Table, Melamine, 30x72*
T8003013	*TRUE*	*Folding Table, Melamine, 30x96*

4.10.7 Manufacturer Item

This example illustrates an import file for reading a manufacturer item data file. See FIGURE 26.

FIGURE 26 Manufacturer Item Import File

```
import_text_file   Item_Manufacturer ()
{
  file: "dist_item.dat, mfg_item.dat";
worksheet ()
{
  model: "Item";

  read Supply_Chain   SUPP = find(supply_chains, "DP Demo");
  read Site           SITE  = find(SUPP.sites, "Manufacturers");

// All manufacturing sub-sites share these items by putting them in the Manufacturers super-site

  this = SITE.items;

  [ A1: name = #; ]                          // Part number for item
  [ B1: "" = #; ]                            // Is item only in whole units
  [ C1: description = #; ]                   // Long description for item

  [ D1: artificial = "FALSE"; ]              // All manufactured items are real items
}
import_record: A1 B1 C1 D1;
}
```

Notes:

1. All manufacturing sub-sites share these items by putting them in the *Manufacturers* super site.
2. Two *read* statements: the first specifies the supply chain name for which data will be read. The second specifies a site name that will also be a super site name.
3. Imports two data files. *name* and *description* for items manufactured at plants as well as end items that are requested and promised (distributor items) are read as items for the *Manufacturers* super *Site* (*this=site.items* indicates the model sub-level) in the *DP Demo* Supply_Chain.
4. Import File variable named SUPP is set to the *DP Demo* Supply_Chain by the first read statement, i.e. SUPP has a value of *DP Demo*.
5. *DP Demo* will have been read as the *name* of a Supply_Chain (by a *Supply_Chain.imp* import file from a *supply_chain.dat* data file).
6. Import File variable named SITE is set to the *Manufacturers* Site by the second read statement.

7. The sites are read into the *DP Demo* Supply_Chain (SUPP).

FIGURE 27 Distributor Item Data File

(End items that are requested, and promised)

name	*discrete*	*description*
C8003001	*TRUE*	*Folding Chair, Steel, Beige*
C8003002	*TRUE*	*Folding Chair, Steel, Brown*
C8003011	*TRUE*	*Folding Chair, Padded, Beige*
C8003012	*TRUE*	*Folding Chair, Padded, Brown*
T8003001	*TRUE*	*Folding Table, Laminate, 30x60*
T8003002	*TRUE*	*Folding Table, Laminate, 30x72*
T8003003	*TRUE*	*Folding Table, Laminate, 30x96*
T8003011	*TRUE*	*Folding Table, Melamine, 30x60*
T8003012	*TRUE*	*Folding Table, Melamine, 30x72*
T8003013	*TRUE*	*Folding Table, Melamine, 30x96*

FIGURE 28 Manufacturer Item Data File

(Items manufactured at plants)

name	*discrete*	*description*
# Chair:Final Assembly		
ASM01000	*TRUE*	*Folding Chair Assembly, Steel*
# Chair:Sub-Assembly		
ASM01200	*TRUE*	*Folding Chair Back Assembly*
ASM01300	*TRUE*	*Folding Chair Leg Assembly*
# Chair:Manufactured Parts		
MFT01100	*TRUE*	*Folding Chair Seat, Steel*
MFT01400	*TRUE*	*Folding Chair U Brace*
MFT01500	*TRUE*	*Folding Chair Backrest, Steel*
MFT01600	*TRUE*	*Folding Chair Back Frame*
MFT01700	*TRUE*	*Folding Chair Legs*
MFT01800	*TRUE*	*Folding Chair Hinge*
# Table:Final Assembly		
ASM05001	*TRUE*	*Folding Table Assembly, 30x60*
ASM05002	*TRUE*	*Folding Table Assembly, 30x72*
ASM05003	*TRUE*	*Folding Table Assembly, 30x96*
# Table:Sub-Assembly		
ASM05101	*TRUE*	*Folding Table Top Assembly, 30x60*
ASM05102	*TRUE*	*Folding Table Top Assembly, 30x72*
ASM05103	*TRUE*	*Folding Table Top Assembly, 30x96*
ASM05300	*TRUE*	*Folding Table Leg Assembly*
ASM05310	*TRUE*	*Folding Table Leg Support Assembly*
# Table:Manufactured Parts		
MFT05400	*TRUE*	*Folding Table Leg Brace*
MFT05700	*TRUE*	*Folding Table Legs*
MFT05800	*TRUE*	*Folding Table Leg Support*
MFT05601	*TRUE*	*Folding Table Rim, 30x60*
MFT05602	*TRUE*	*Folding Table Rim, 30x72*
MFT05603	*TRUE*	*Folding Table Rim, 30x96*
# Raw Material		
MAT20000	*FALSE*	*Sheet Steel, .15 in*
MAT20100	*FALSE*	*Strip Steel, .15 in*
MAT30010	*FALSE*	*Steel Tubing, .4 in*
MAT30020	*FALSE*	*Steel Tubing, 1 in*
MAT30021	*FALSE*	*Steel Tubing, .6 in*
MAT40010	*TRUE*	*Steel Rivet, #10, 0.37 in*
MAT40020	*TRUE*	*Steel Rivet, #8, 0.75 in*
MAT40110	*TRUE*	*Steel Screw, #10, 0.37 in*
MAT40120	*TRUE*	*Steel Screw, #8, 0.75 in*
MAT40041	*TRUE*	*Steel U Mounting Bracket*
MAT40042	*TRUE*	*Steel L Mounting Bracket*
MAT80101	*TRUE*	*Rubber Foot, Beige, 1 in*
MAT80102	*TRUE*	*Rubber Foot, Brown, 1 in*
MAT80201	*TRUE*	*Rubber End, Beige, 1 in*

4.10.8 Market Area Item

This example illustrates an import file for reading a market area item data file. See FIGURE 29.

FIGURE 29 Market Area Item Import File

```
import_text_file   Item_Market_Area ()
{
  file: "dist_item.dat";
worksheet ()
{
  model:  "Item";

  read Supply_Chain   SUPP = find(supply_chains, "DP Demo");
  read Site           SITE  = find(SUPP.sites,   "Market Areas");

// All market area sub-sites share these items by putting them in the "Market Areas" super-site

  this = SITE.items;

  [ A1: name = #; ]                          // Part number for item
  [ B1: "" = #; ]                            // Is item only in whole units
  [ C1: description = #; ]                   // Long description for item

  [ D1: artificial = "FALSE"; ]              // Is this a real item
}
import_record: A1 B1 C1 D1;
}
```

Notes:

1. All market area sub-sites share these items by putting them in the "Market Areas" super-site.
2. Imports distributor items (end items that are requested, and promised).

FIGURE 30 Distributor Item Data File

(End items that are requested, and promised)

name	*discrete*	*description*
C8003001	*TRUE*	*Folding Chair, Steel, Beige*
C8003002	*TRUE*	*Folding Chair, Steel, Brown*
C8003011	*TRUE*	*Folding Chair, Padded, Beige*
C8003012	*TRUE*	*Folding Chair, Padded, Brown*
T8003001	*TRUE*	*Folding Table, Laminate, 30x60*
T8003002	*TRUE*	*Folding Table, Laminate, 30x72*
T8003003	*TRUE*	*Folding Table, Laminate, 30x96*
T8003011	*TRUE*	*Folding Table, Melamine, 30x60*
T8003012	*TRUE*	*Folding Table, Melamine, 30x72*
T8003013	*TRUE*	*Folding Table, Melamine, 30x96*

4.10.9 Distributor Operation

This example illustrates an import file for reading a distributor operation data file. See FIGURE 31.

FIGURE 31 Distributor Operation Import File

```
import_text_file   Operation_Distributor ()
{
  file: "dist_operation.dat";
worksheet ()
{
  model:  "Operation";

  read Supply_Chain   SUPP = find(supply_chains, "DP Demo");

  this = SUPP.sites;

  [ A1: name = #; ]                                  // Site where operation resides
  [ B1: operations.name = #; ]                       // Short name for operation
  [ C1: operations.description = #; ]                // Long description for operation

  [ D1: operations.process  = "ALTERNATES_PRIMARY"; ] // Distributor operations have alternates
}
import_record: A1 B1 C1 D1;
}
```


FIGURE 32 Distributor Operation Data File

site	*item*	*description*
# Houston, TX Warehouse		
HOU	*GET-C8003001*	*Get C8003001 for Houston, TX Warehouse*
HOU	*GET-C8003002*	*Get C8003002 for Houston, TX Warehouse*
# Los Angeles, CA Warehouse		
LOS	*GET-C8003001*	*Get C8003001 for Los Angeles, CA Warehouse*
LOS	*GET-C8003002*	*Get C8003002 for Los Angeles, CA Warehouse*
# London, England Warehouse		
LON	*GET-C8003001*	*Get C8003001 for London, England Warehouse*
LON	*GET-C8003002*	*Get C8003002 for London, England Warehouse*
# New York, NY Warehouse		
NEW	*GET-C8003001*	*Get C8003001 for New York, NY Warehouse*
NEW	*GET-C8003002*	*Get C8003002 for New York, NY Warehouse*
# European Market Area		
EUR	*GET-C8003001*	*Get C8003001 for European Market Area*
EUR	*GET-C8003002*	*Get C8003002 for European Market Area*
# Japanese Market Area		
JAP	*GET-C8003001*	*Get C8003001 for Japanese Market Area*
JAP	*GET-C8003002*	*Get C8003002 for Japanese Market Area*
# Midwestern US Market Area		
MID	*GET-C8003001*	*Get C8003001 for Midwestern US Market Area*
MID	*GET-C8003002*	*Get C8003002 for Midwestern US Market Area*
# Northern US Market Area		
NOR	*GET-C8003001*	*Get C8003001 for Northern US Market Area*
NOR	*GET-C8003002*	*Get C8003002 for Northern US Market Area*
# Southern US Market Area		
SOU	*GET-C8003001*	*Get C8003001 for Southern US Market Area*
SOU	*GET-C8003002*	*Get C8003002 for Southern US Market Area*
# Southwestern US Market Area		
SWS	*GET-C8003001*	*Get C8003001 for Southwestern US Market Area*
SWS	*GET-C8003002*	*Get C8003002 for Southwestern US Market Area*
# Western US Market Area		
WST	*GET-C8003001*	*Get C8003001 for Western US Market Area*
WST	*GET-C8003002*	*Get C8003002 for Western US Market Area*
# California Market Area		
CA	*GET-C8003001*	*Get C8003001 for California Market Area*
CA	*GET-C8003002*	*Get C8003002 for California Market Area*
# New York Market Area		
NY	*GET-C8003001*	*Get C8003001 for New York Market Area*
NY	*GET-C8003002*	*Get C8003002 for New York Market Area*
# Ohio Market Area		
OH	*GET-C8003001*	*Get C8003001 for Ohio Market Area*
OH	*GET-C8003002*	*Get C8003002 for Ohio Market Area*
# Texas Market Area		
TX	*GET-C8003001*	*Get C8003001 for Texas Market Area*
TX	*GET-C8003002*	*Get C8003002 for Texas Market Area*

4.10.10 Distributor Alternate Operation

This example illustrates an import file for reading a distributor alternate operation data file. See FIGURE 33.

FIGURE 33 Distributor Alternate Operation Import File

```
import_text_file   Operation_Distributor_Alternate ()
{
  file: "dist_operation_req.dat";
worksheet ()
{
  model:  "Alternate_Operation";

  read Supply_Chain   SUPP = find(supply_chains, "DP Demo");

  this = SUPP.sites;

  [ A1: name = #; ]                                                    // Site where operation resides
  [ B1 = operations.name = #,          = "GET-" & #; ]                 // Short name for parent operation
  [ C1: operations.alternates.operation, = "GET-" & B1 & "-FROM-" & #; ] // Short name for alternate operation
}
import_record: A1 B1 C1;
}
```


FIGURE 34 Distributor Alternate Operation Data File

```
site    item        from   time     description

# Houston, TX Warehouse
# Boston, MA Plant
HOU     C8003001    BOS    4 day    Move C8003001 to Houston, TX Warehouse from Boston, MA Plant
HOU     C8003002    BOS    4 day    Move C8003002 to Houston, TX Warehouse from Boston, MA Plant
# Houston, TX Warehouse
# Dallas, TX Plant
HOU     C8003001    DAL    1 day    Move C8003001 to Houston, TX Warehouse from Dallas, TX Plant
HOU     C8003002    DAL    1 day    Move C8003002 to Houston, TX Warehouse from Dallas, TX Plant
# Houston, TX Warehouse
# Riverside, CA Plant
HOU     C8003001    RIV    4 day    Move C8003001 to Houston, TX Warehouse from Riverside, CA Plant
HOU     C8003002    RIV    4 day    Move C8003002 to Houston, TX Warehouse from Riverside, CA Plant
# Japanese Market Area
# Los Angeles, CA Warehouse
JAP     C8003001    LOS    7 day    Move C8003001 to Japanese Market Area from LOS
JAP     C8003002    LOS    7 day    Move C8003002 to Japanese Market Area from LOS
# Midwestern US Market Area
# Houston, TX Warehouse
MID     C8003001    HOU    3 day    Move C8003001 to Midwestern US Market Area from HOU
MID     C8003002    HOU    3 day    Move C8003002 to Midwestern US Market Area from HOU
# Northern US Market Area
# Houston, TX Warehouse
NOR     C8003001    HOU    3 day    Move C8003001 to Northern US Market Area from HOU
NOR     C8003002    HOU    3 day    Move C8003002 to Northern US Market Area from HOU
# Southern US Market Area
# Houston, TX Warehouse
SOU     C8003001    HOU    2 day    Move C8003001 to Southern US Market Area from HOU
SOU     C8003002    HOU    2 day    Move C8003002 to Southern US Market Area from HOU
# Southwestern US Market Area
# Houston, TX Warehouse
SWS     C8003001    HOU    2 day    Move C8003001 to Southwestern US Market Area from HOU
SWS     C8003002    HOU    2 day    Move C8003002 to Southwestern US Market Area from HOU
# New York Market Area
# Houston, TX Warehouse
NY      C8003001    HOU    4 day    Move C8003001 to New York Market Area from HOU
NY      C8003002    HOU    4 day    Move C8003002 to New York Market Area from HOU
NY      C8003011    HOU    4 day    Move C8003011 to New York Market Area from HOU
# Houston, TX Warehouse
OH      C8003001    HOU    3 day    Move C8003001 to Ohio Market Area from HOU
OH      C8003002    HOU    3 day    Move C8003002 to Ohio Market Area from HOU
# Texas Market Area
# Houston, TX Warehouse
TX      C8003001    HOU    1 day    Move C8003001 to Texas Market Area from HOU
TX      C8003002    HOU    1 day    Move C8003002 to Texas Market Area from HOU
```

4.10.11 Distributor Delivery Operation

This example illustrates an import file for reading a distributor delivery operation data file. See FIGURE 35.

FIGURE 35 Distributor Delivery Operation Import File

```
import_text_file   Operation_Distributor_Delivery ()
{
  file: "dist_operation_del.dat";
worksheet ()
{
  model: "Operation";

  read Supply_Chain   SUPP = find(supply_chains, "DP Demo");

  this = SUPP.sites;

  [ A1: name = #; ]                                    // Site where operation resides
  [ B1: operations.name = #, = "DELIVER-" & #; ]       // Short name for operation
  [ C1: operations.time = #; ]                         // Time that operation requires
  [ D1: operations.description = #; ]                  // Long description for operation

  [ E1: operations.flows.buffer    = B1; ]             // Buffer from which distributor operation delivers
  [ F1: operations.flows.usage_policy= "CONSUME_FIXED"; ] // Distributor operations empty buffers
  [G1: operations.process          = "FIXED_TIME"; ]   // Distributor delivery operations take fixed time
}
import_record: A1 B1 C1 D1 E1 F1 G1;
}
```


FIGURE 36 Distributor Delivery Operation Data File

site	*item*	*time*	*description*
# Houston, TX Warehouse			
HOU	*C8003001*	*2 hour*	*Deliver C8003001 from Houston, TX Warehouse*
HOU	*C8003002*	*2 hour*	*Deliver C8003002 from Houston, TX Warehouse*
LOS	*C8003001*	*2 hour*	*Deliver C8003001 from Los Angeles, CA Warehouse*
LOS	*C8003002*	*2 hour*	*Deliver C8003002 from Los Angeles, CA Warehouse*
# London, England Warehouse			
LON	*C8003001*	*2 hour*	*Deliver C8003001 from London, England Warehouse*
LON	*C8003002*	*2 hour*	*Deliver C8003002 from London, England Warehouse*
# New York, NY Warehouse			
NEW	*C8003001*	*2 hour*	*Deliver C8003001 from New York, NY Warehouse*
NEW	*C8003002*	*2 hour*	*Deliver C8003002 from New York, NY Warehouse*
# European Market Area			
EUR	*C8003001*	*2 hour*	*Deliver C8003001 from European Market Area*
EUR	*C8003002*	*2 hour*	*Deliver C8003002 from European Market Area*
# Japanese Market Area			
JAP	*C8003001*	*2 hour*	*Deliver C8003001 from Japanese Market Area*
JAP	*C8003002*	*2 hour*	*Deliver C8003002 from Japanese Market Area*
# Midwestern US Market Area			
MID	*C8003001*	*2 hour*	*Deliver C8003001 from Midwestern US Market Area*
MID	*C8003002*	*2 hour*	*Deliver C8003002 from Midwestern US Market Area*
# Northern US Market Area			
NOR	*C8003001*	*2 hour*	*Deliver C8003001 from Northern US Market Area*
NOR	*C8003002*	*2 hour*	*Deliver C8003002 from Northern US Market Area*
# Southern US Market Area			
SOU	*C8003001*	*2 hour*	*Deliver C8003001 from Southern US Market Area*
SOU	*C8003002*	*2 hour*	*Deliver C8003002 from Southern US Market Area*
# Southwestern US Market Area			
SWS	*C8003001*	*2 hour*	*Deliver C8003001 from Southwestern US Market Area*
SWS	*C8003002*	*2 hour*	*Deliver C8003002 from Southwestern US Market Area*
# Western US Market Area			
WST	*C8003001*	*2 hour*	*Deliver C8003001 from Western US Market Area*
WST	*C8003002*	*2 hour*	*Deliver C8003002 from Western US Market Area*
# California Market Area			
CA	*C8003001*	*2 hour*	*Deliver C8003001 from California Market Area*
CA	*C8003002*	*2 hour*	*Deliver C8003002 from California Market Area*
# New York Market Area			
NY	*C8003001*	*2 hour*	*Deliver C8003001 from New York Market Area*
NY	*C8003002*	*2 hour*	*Deliver C8003002 from New York Market Area*
# Ohio Market Area			
OH	*C8003001*	*2 hour*	*Deliver C8003001 from Ohio Market Area*
OH	*C8003002*	*2 hour*	*Deliver C8003002 from Ohio Market Area*
# Texas Market Area			
TX	*C8003001*	*2 hour*	*Deliver C8003001 from Texas Market Area*
TX	*C8003002*	*2 hour*	*Deliver C8003002 from Texas Market Area*

4.10.12 Manufacturer Operation

This example illustrates an import file for reading a manufacturer operation data file. See FIGURE 37.

FIGURE 37 Manufacturer Operation Import File

```
import_text_file   Operation_Manufacturer ()
{
  file: "mfg_operation.dat";
worksheet ()
{
  model:  "Operation";

  read Supply_Chain   SUPP = find(supply_chains, "DP Demo");

  this = SUPP.sites;

  [ A1: name = #; ]                                  // Site for operation
  [ B1: operations.name = #; ]                       // Short name for operation
  [ C1: "" = #; ]                                    // Does operation produce whole units
  [ D1: operations.time = #; ]                       // Time that operation requires
  [ E1: operations.description = #; ]                // Long description for operation

  [ F1: operations.process = "FIXED_TIME"; ]         // Operation takes fixed time
}
import_record: A1 B1 C1 D1 E1 F1;
}
```


FIGURE 38 Manufacturer Operation Data File

site	*name*	*discrete*	*time*	*description*
# Boston, MA Plant				
BOS	*DELIVER-C8003001*	*TRUE*	*2 hour*	*Deliver C8003001 from Boston, MA Plant*
BOS	*DELIVER-C8003002*	*TRUE*	*2 hour*	*Deliver C8003002 from Boston, MA Plant*
BOS	*DELIVER-C8003011*	*TRUE*	*2 hour*	*Deliver C8003011 from Boston, MA Plant*
BOS	*DELIVER-C8003012*	*TRUE*	*2 hour*	*Deliver C8003012 from Boston, MA Plant*
BOS	*DELIVER-T8003001*	*TRUE*	*2 hour*	*Deliver T8003001 from Boston, MA Plant*
BOS	*DELIVER-T8003002*	*TRUE*	*2 hour*	*Deliver T8003002 from Boston, MA Plant*
BOS	*DELIVER-T8003003*	*TRUE*	*2 hour*	*Deliver T8003003 from Boston, MA Plant*
BOS	*DELIVER-T8003011*	*TRUE*	*2 hour*	*Deliver T8003011 from Boston, MA Plant*
BOS	*DELIVER-T8003012*	*TRUE*	*2 hour*	*Deliver T8003012 from Boston, MA Plant*
BOS	*DELIVER-T8003013*	*TRUE*	*2 hour*	*Deliver T8003013 from Boston, MA Plant*
BOS	*MAKE-C8003001*	*TRUE*	*2 day*	*Make C8003001 at Boston, MA Plant*
BOS	*MAKE-C8003002*	*TRUE*	*2 day*	*Make C8003002 at Boston, MA Plant*
BOS	*MAKE-C8003011*	*TRUE*	*2 day*	*Make C8003011 at Boston, MA Plant*
BOS	*MAKE-C8003012*	*TRUE*	*2 day*	*Make C8003012 at Boston, MA Plant*
# Dallas, TX Plant				
DAL	*DELIVER-C8003001*	*TRUE*	*2 hour*	*Deliver C8003001 from Dallas, TX Plant*
DAL	*DELIVER-C8003002*	*TRUE*	*2 hour*	*Deliver C8003002 from Dallas, TX Plant*
DAL	*DELIVER-C8003011*	*TRUE*	*2 hour*	*Deliver C8003011 from Dallas, TX Plant*
DAL	*DELIVER-C8003012*	*TRUE*	*2 hour*	*Deliver C8003012 from Dallas, TX Plant*
DAL	*DELIVER-T8003001*	*TRUE*	*2 hour*	*Deliver T8003001 from Dallas, TX Plant*
DAL	*DELIVER-T8003002*	*TRUE*	*2 hour*	*Deliver T8003002 from Dallas, TX Plant*
DAL	*DELIVER-T8003003*	*TRUE*	*2 hour*	*Deliver T8003003 from Dallas, TX Plant*
DAL	*DELIVER-T8003011*	*TRUE*	*2 hour*	*Deliver T8003011 from Dallas, TX Plant*
DAL	*DELIVER-T8003012*	*TRUE*	*2 hour*	*Deliver T8003012 from Dallas, TX Plant*
DAL	*DELIVER-T8003013*	*TRUE*	*2 hour*	*Deliver T8003013 from Dallas, TX Plant*
DAL	*MAKE-C8003001*	*TRUE*	*2 day*	*Make C8003001 at Dallas, TX Plant*
DAL	*MAKE-C8003002*	*TRUE*	*2 day*	*Make C8003002 at Dallas, TX Plant*
DAL	*MAKE-C8003011*	*TRUE*	*2 day*	*Make C8003011 at Dallas, TX Plant*
DAL	*MAKE-C8003012*	*TRUE*	*2 day*	*Make C8003012 at Dallas, TX Plant*
DAL	*MAKE-T8003001*	*TRUE*	*2 day*	*Make T8003001 at Dallas, TX Plant*
DAL	*MAKE-T8003002*	*TRUE*	*2 day*	*Make T8003002 at Dallas, TX Plant*
DAL	*MAKE-T8003003*	*TRUE*	*2 day*	*Make T8003003 at Dallas, TX Plant*
DAL	*MAKE-T8003011*	*TRUE*	*2 day*	*Make T8003011 at Dallas, TX Plant*
DAL	*MAKE-T8003012*	*TRUE*	*2 day*	*Make T8003012 at Dallas, TX Plant*
# Riverside, CA Plant				
RIV	*DELIVER-C8003001*	*TRUE*	*2 hour*	*Deliver C8003001 from Riverside, CA Plant*
RIV	*DELIVER-C8003002*	*TRUE*	*2 hour*	*Deliver C8003002 from Riverside, CA Plant*
RIV	*DELIVER-C8003011*	*TRUE*	*2 hour*	*Deliver C8003011 from Riverside, CA Plant*
RIV	*DELIVER-C8003012*	*TRUE*	*2 hour*	*Deliver C8003012 from Riverside, CA Plant*
RIV	*DELIVER-T8003001*	*TRUE*	*2 hour*	*Deliver T8003001 from Riverside, CA Plant*
RIV	*DELIVER-T8003002*	*TRUE*	*2 hour*	*Deliver T8003002 from Riverside, CA Plant*
RIV	*DELIVER-T8003003*	*TRUE*	*2 hour*	*Deliver T8003003 from Riverside, CA Plant*
RIV	*DELIVER-T8003011*	*TRUE*	*2 hour*	*Deliver T8003011 from Riverside, CA Plant*
RIV	*DELIVER-T8003012*	*TRUE*	*2 hour*	*Deliver T8003012 from Riverside, CA Plant*

4.10.13 Manufacturer Operation Flow

This example illustrates an import file for reading a manufacturer operation flow data file. See FIGURE 39.

FIGURE 39 Manufacturer Operation Flow Import File

```
import_text_file   Operation_Manufacturer_Flow ()
{
  file: "mfg_operation_flow.dat";
worksheet ()
{
  model: "Flow";

  read Supply_Chain   SUPP = find(supply_chains, "DP Demo");

  this = SUPP.sites;

  [ A1: name = #; ]                          // Site for operation
  [ B1: operations.name = #; ]               // Owning operation for flow
  [ C1: operations.flows.buffer = #; ]       // Buffer that flow uses
  [ D1: operations.flows.usage_policy = #; ] // What use flow makes of the buffer
}
import_record: A1 B1 C1 D1;
}
```


FIGURE 40 Manufacturer Operation Flow Data File

site	*name*	*buffer*	*usage policy*	*qty*
#Boston, MA Plant				
BOS	*MAKE-C8003001*	*C8003001*	*PRODUCE_FIXED*	*1*
BOS	*MAKE-C8003002*	*C8003002*	*PRODUCE_FIXED*	*1*
BOS	*MAKE-C8003011*	*C8003011*	*PRODUCE_FIXED*	*1*
BOS	*MAKE-C8003012*	*C8003012*	*PRODUCE_FIXED*	*1*
BOS	*MAKE-T8003001*	*T8003001*	*PRODUCE_FIXED*	*1*
BOS	*MAKE-T8003002*	*T8003002*	*PRODUCE_FIXED*	*1*
BOS	*MAKE-T8003003*	*T8003003*	*PRODUCE_FIXED*	*1*
BOS	*MAKE-T8003011*	*T8003011*	*PRODUCE_FIXED*	*1*
BOS	*MAKE-T8003012*	*T8003012*	*PRODUCE_FIXED*	*1*
#Dallas, TX Plant				
DAL	*MAKE-C8003001*	*C8003001*	*PRODUCE_FIXED*	*1*
DAL	*MAKE-C8003002*	*C8003002*	*PRODUCE_FIXED*	*1*
DAL	*MAKE-C8003011*	*C8003011*	*PRODUCE_FIXED*	*1*
DAL	*MAKE-C8003012*	*C8003012*	*PRODUCE_FIXED*	*1*
DAL	*MAKE-T8003001*	*T8003001*	*PRODUCE_FIXED*	*1*
DAL	*MAKE-T8003002*	*T8003002*	*PRODUCE_FIXED*	*1*
DAL	*MAKE-T8003003*	*T8003003*	*PRODUCE_FIXED*	*1*
DAL	*MAKE-T8003011*	*T8003011*	*PRODUCE_FIXED*	*1*
DAL	*MAKE-T8003012*	*T8003012*	*PRODUCE_FIXED*	*1*
#Riverside, CA Plant				
RIV	*MAKE-C8003001*	*C8003001*	*PRODUCE_FIXED*	*1*
RIV	*MAKE-C8003002*	*C8003002*	*PRODUCE_FIXED*	*1*
RIV	*MAKE-C8003011*	*C8003011*	*PRODUCE_FIXED*	*1*
RIV	*MAKE-C8003012*	*C8003012*	*PRODUCE_FIXED*	*1*
RIV	*MAKE-T8003001*	*T8003001*	*PRODUCE_FIXED*	*1*
RIV	*MAKE-T8003002*	*T8003002*	*PRODUCE_FIXED*	*1*
RIV	*MAKE-T8003003*	*T8003003*	*PRODUCE_FIXED*	*1*
RIV	*MAKE-T8003011*	*T8003011*	*PRODUCE_FIXED*	*1*
RIV	*MAKE-T8003012*	*T8003012*	*PRODUCE_FIXED*	*1*

4.10.14 Plan

This example illustrates an import file for reading a plan data file. See FIGURE 41.

FIGURE 41 Plan Import File

```
import_text_file   Plan ()
{
  file: "plan.dat";
worksheet ()
{
  model: "Plan";

  read Supply_Chain   SUPP = find(supply_chains, "DP Demo");

  this = SUPP.plans;

  [ A1: name = #; ]                    // Short name for the plan
  [ B1: period = #; ]                  // Period of time the plan covers
  [ C1: description = #; ]             // Long description for the plan
}
import_record: A1 B1 C1;
}
```


FIGURE 42 Plan Data File

name	*period*	*description*
Active	*07/01/95 00:00:00 / 01/01/96 00:00:00*	*Active plan currently being executed*
New WH	*07/01/95 00:00:00 / 01/01/96 00:00:00*	*What if plan with new warehouse added*
FY96	*10/01/95 00:00:00 / 10/01/96 00:00:00*	*Long range plan for 1996 Fiscal Year*

4.10.15 Product

This example illustrates an import file for reading a product data file. See FIGURE 43.

FIGURE 43 Product Import File

```
import_text_file   Product ()
{
  file: "product.dat";
worksheet ()
{
  model:  "Product";

  read Supply_Chain   SUPP = find(supply_chains, "DP Demo");

  this = SUPP.sellers;

  [ A1: name = #; ]                          // Owning seller for Product
  [ B1: products.name = #; ]                 // Short name for Product
  [ C1: products.description = #; ]          // Long description for Product

  [ D1: products.forecast_policy = "SIMPLE"; ] // Use standard forecasting approach
  [ E1: products.items.item   = B1; ]        // Item this product represents
  [ F1: products.supplier   = A1; ]          // Supplying site for item
}
import_record: A1 B1 C1 D1 E1 F1;
}
```


FIGURE 44 Product Data File

site	*item*	*description*
#Central US Market Area		
CEN	*C8003001*	*Product for item C8003001 at CEN*
CEN	*C8003002*	*Product for item C8003002 at CEN*
#Eastern US Market Area		
EAS	*C8003001*	*Product for item C8003001 at EAS*
EAS	*C8003002*	*Product for item C8003002 at EAS*
#European Market Area		
EUR	*C8003001*	*Product for item C8003001 at EUR*
EUR	*C8003002*	*Product for item C8003002 at EUR*
#Japanese Market Area		
JAP	*C8003001*	*Product for item C8003001 at JAP*
JAP	*C8003002*	*Product for item C8003002 at JAP*
#Midwestern US Market Area		
MID	*C8003001*	*Product for item C8003001 at MID*
MID	*C8003002*	*Product for item C8003002 at MID*
#Northern US Market Area		
NOR	*C8003001*	*Product for item C8003001 at NOR*
NOR	*C8003002*	*Product for item C8003002 at NOR*
#Southern US Market Area		
SOU	*C8003001*	*Product for item C8003001 at SOU*
SOU	*C8003002*	*Product for item C8003002 at SOU*
#Southwestern US Market Area		
SWS	*C8003001*	*Product for item C8003001 at SWS*
SWS	*C8003002*	*Product for item C8003002 at SWS*
#Western US Market Area		
WST	*C8003001*	*Product for item C8003001 at WST*
WST	*C8003002*	*Product for item C8003002 at WST*
#California Market Area		
CA	*C8003001*	*Product for item C8003001 at CA*
CA	*C8003002*	*Product for item C8003002 at CA*
#New York Market Area		
NY	*C8003001*	*Product for item C8003001 at NY*
NY	*C8003002*	*Product for item C8003002 at NY*
#Ohio Market Area		
OH	*C8003001*	*Product for item C8003001 at OH*
OH	*C8003002*	*Product for item C8003002 at OH*
#Texas Market Area		
TX	*C8003001*	*Product for item C8003001 at TX*
TX	*C8003002*	*Product for item C8003002 at TX*

4.10.16 Product Group

This example illustrates an import file for reading a product group data file. See FIGURE 45.

FIGURE 45 Product Group Import File

```
import_text_file   Product_Group ()
{
  file: "product_group.dat";
worksheet ()
{
  model:  "Product_Group";

  read Supply_Chain   SUPP = find(supply_chains, "DP Demo");

  this = SUPP.sellers;

  [ A1: .name = #; ]                          // Owning seller for Product
  [ B1: products_groups.name = #; ]           // Short name for Product Group
  [ C1: products_groups.description = #; ]    // Long description for Product Group
}
import_record: A1 B1 C1;
}
```


FIGURE 46 Product Group Data File

(product groups for products that are requested, and promised)

seller	*name*	*description*
#Central US Market Area		
CEN	*CHA*	*Product group for Chair at CEN*
CEN	*STE*	*Product group for Folding Chair, Steel at CEN*
CEN	*PAD*	*Product group for Folding Chair, Padded at CEN*
CEN	*LAM*	*Product group for Folding Table, Laminated at CEN*
CEN	*MEL*	*Product group for Folding Table, Melamine at CEN*
#European Market Area		
EUR	*CHA*	*Product group for Chair at EUR*
EUR	*TAB*	*Product group for Folding Table at EUR*
EUR	*STE*	*Product group for Folding Chair, Steel at EUR*
EUR	*PAD*	*Product group for Folding Chair, Padded at EUR*
EUR	*LAM*	*Product group for Folding Table, Laminated at EUR*
EUR	*MEL*	*Product group for Folding Table, Melamine at EUR*
#Japanese Market Area		
JAP	*CHA*	*Product group for Chair at JAP*
JAP	*TAB*	*Product group for Folding Table at JAP*
#Midwestern US Market Area		
MID	*CHA*	*Product group for Chair at MID*
#Northern US Market Area		
NOR	*CHA*	*Product group for Chair at NOR*
NOR	*TAB*	*Product group for Folding Table at NOR*
#Southern US Market Area		
SOU	*CHA*	*Product group for Chair at SOU*
SOU	*TAB*	*Product group for Folding Table at SOU*
#Southwestern US Market Area		
SWS	*CHA*	*Product group for Chair at SWS*
SWS	*TAB*	*Product group for Folding Table at SWS*
#Western US Market Area		
WST	*CHA*	*Product group for Chair at WST*
WST	*TAB*	*Product group for Folding Table at WST*
#California Market Area		
CA	*CHA*	*Product group for Chair at CA*
CA	*TAB*	*Product group for Folding Table at CA*
#New York Market Area		
NY	*CHA*	*Product group for Chair at NY*
NY	*TAB*	*Product group for Folding Table at NY*
#Ohio Market Area		
OH	*CHA*	*Product group for Chair at OH*
OH	*TAB*	*Product group for Folding Table at OH*
#Texas Market Area		
TX	*CHA*	*Product group for Chair at TX*
TX	*TAB*	*Product group for Folding Table at TX*

4.10.17 Request

This example illustrates an import file for reading a request data file. See FIGURE 47.

FIGURE 47 Request Import File

```
import_text_file   Request ()
{
  file: "request.dat";
worksheet ()
{
  model: "Request";

  read Supply_Chain   SUPP   = find(supply_chains, "DP Demo");
  read Plan           PLAN   = find(SUPP.plans, "Active");

  this = PLAN.site_plans;

  [ A1: site = #; ]                                          // Site serving the request
  [ B1 = requests.name = #; ]                                // Short name for the request
  [ C1 = requests.order_requests.item_requests.name = #; ]   // Short name for the item request
  [ D1: requests.order_requests.item_requests.min_quantity = #; ] // Quantity requested
  [ E1: requests.order_requests.due = #; ]                   // Due date for the order request
  [ F1: requests.description = #; ]                          // Long description for the request

  [ G1: requests.seller_plan.seller = A1; ]                  // Seller plan to fill this request

  [ H1: requests.order_requests.name = B1; ]                 // Short name for the order request

  [ I1: requests.order_requests.item_requests.item   = C1; ] // Item requested
  [ J1: requests.order_requests.item_requests.max_quantity = D1; ] // Quantity requested
}
import_record: A1 B1 C1 D1 E1 F1 G1 H1 I1 J1;
}
```


FIGURE 48 Request Data File

site	*name*	*item*	*qty*	*due date period*	*description*
CEN	*ORDER-001*	*C8003002*	*20*	*07/11/95 00:00:00 / 07/11/95 20:00:00*	*Order for item C8003002 from CEN*
CEN	*ORDER-002*	*C8003001*	*20*	*08/23/95 00:00:00 / 08/23/95 20:00:00*	*Order for item C8003001 from CEN*
EAS	*ORDER-013*	*C8003001*	*20*	*07/15/95 00:00:00 / 07/15/95 20:00:00*	*Order for item C8003001 from EAS*
EAS	*ORDER-014*	*C8003001*	*20*	*08/04/95 00:00:00 / 08/04/95 20:00:00*	*Order for item C8003001 from EAS*
EAS	*ORDER-024*	*T8003002*	*10*	*12/01/95 00:00:00 / 12/01/96 20:00:00*	*Order for item T8003002 from EAS*
EUR	*ORDER-025*	*C8003012*	*20*	*07/12/95 00:00:00 / 07/12/95 20:00:00*	*Order for item C8003012 from EUR*
EUR	*ORDER-026*	*C8003001*	*20*	*08/18/95 00:00:00 / 08/18/95 20:00:00*	*Order for item C8003001 from EUR*
EUR	*ORDER-036*	*T8003001*	*10*	*12/02/95 00:00:00 / 12/02/96 20:00:00*	*Order for item T8003001 from EUR*
JAP	*ORDER-037*	*C8003002*	*20*	*07/26/95 00:00:00 / 07/26/95 20:00:00*	*Order for item C8003002 from JAP*
JAP	*ORDER-038*	*C8003002*	*20*	*08/11/95 00:00:00 / 08/11/95 20:00:00*	*Order for item C8003002 from JAP*
JAP	*ORDER-048*	*T8003003*	*10*	*12/20/95 00:00:00 / 12/20/96 20:00:00*	*Order for item T8003003 from JAP*
MID	*ORDER-049*	*C8003012*	*20*	*07/30/95 00:00:00 / 07/30/95 20:00:00*	*Order for item C8003012 from MID*
MID	*ORDER-050*	*C8003012*	*20*	*08/16/95 00:00:00 / 08/16/95 20:00:00*	*Order for item C8003012 from MID*
MID	*ORDER-060*	*T8003012*	*10*	*12/03/95 00:00:00 / 12/03/96 20:00:00*	*Order for item T8003012 from MID*
NOR	*ORDER-061*	*C8003011*	*20*	*07/29/95 00:00:00 / 07/29/95 20:00:00*	*Order for item C8003011 from NOR*
NOR	*ORDER-062*	*C8003001*	*20*	*08/09/95 00:00:00 / 08/09/95 20:00:00*	*Order for item C8003001 from NOR*
NOR	*ORDER-072*	*T8003001*	*10*	*12/16/95 00:00:00 / 12/16/96 20:00:00*	*Order for item T8003001 from NOR*
SOU	*ORDER-073*	*C8003001*	*20*	*07/04/95 00:00:00 / 07/04/95 20:00:00*	*Order for item C8003001 from SOU*
SOU	*ORDER-074*	*C8003012*	*20*	*08/21/95 00:00:00 / 08/21/95 20:00:00*	*Order for item C8003012 from SOU*
SOU	*ORDER-084*	*T8003011*	*10*	*12/14/95 00:00:00 / 12/14/96 20:00:00*	*Order for item T8003011 from SOU*
SWS	*ORDER-085*	*C8003011*	*20*	*07/16/95 00:00:00 / 07/16/95 20:00:00*	*Order for item C8003011 from SWS*
SWS	*ORDER-086*	*C8003012*	*20*	*08/18/95 00:00:00 / 08/18/95 20:00:00*	*Order for item C8003012 from SWS*
SWS	*ORDER-096*	*T8003002*	*10*	*12/09/95 00:00:00 / 12/09/96 20:00:00*	*Order for item T8003002 from SWS*
WST	*ORDER-097*	*C8003002*	*20*	*07/11/95 00:00:00 / 07/11/95 20:00:00*	*Order for item C8003002 from WST*
WST	*ORDER-098*	*C8003012*	*20*	*08/13/95 00:00:00 / 08/13/95 20:00:00*	*Order for item C8003012 from WST*
WST	*ORDER-108*	*T8003013*	*10*	*12/01/95 00:00:00 / 12/01/96 20:00:00*	*Order for item T8003013 from WST*
CA	*ORDER-109*	*C8003002*	*20*	*07/09/95 00:00:00 / 07/09/95 20:00:00*	*Order for item C8003002 from CA*
CA	*ORDER-110*	*C8003011*	*20*	*08/07/95 00:00:00 / 08/07/95 20:00:00*	*Order for item C8003011 from CA*
CA	*ORDER-120*	*T8003013*	*10*	*12/12/95 00:00:00 / 12/12/96 20:00:00*	*Order for item T8003013 from CA*
NY	*ORDER-121*	*C8003002*	*20*	*07/13/95 00:00:00 / 07/13/95 20:00:00*	*Order for item C8003002 from NY*
NY	*ORDER-122*	*C8003011*	*20*	*08/14/95 00:00:00 / 08/14/95 20:00:00*	*Order for item C8003011 from NY*
NY	*ORDER-132*	*T8003012*	*10*	*12/01/95 00:00:00 / 12/01/96 20:00:00*	*Order for item T8003012 from NY*
OH	*ORDER-133*	*C8003012*	*20*	*07/02/95 00:00:00 / 07/02/95 20:00:00*	*Order for item C8003012 from OH*
OH	*ORDER-134*	*C8003001*	*20*	*08/03/95 00:00:00 / 08/03/95 20:00:00*	*Order for item C8003001 from OH*
OH	*ORDER-144*	*T8003011*	*10*	*12/13/95 00:00:00 / 12/13/96 20:00:00*	*Order for item T8003011 from OH*
TX	*ORDER-145*	*C8003002*	*20*	*07/14/95 00:00:00 / 07/14/95 20:00:00*	*Order for item C8003002 from TX*
TX	*ORDER-146*	*C8003011*	*20*	*08/15/95 00:00:00 / 08/15/95 20:00:00*	*Order for item C8003011 from TX*
TX	*ORDER-156*	*T8003011*	*10*	*12/01/95 00:00:00 / 12/01/96 20:00:00*	*Order for item T8003011 from TX*

4.10.18 Seller

This example illustrates an import file for reading a seller data file. See FIGURE 49.

FIGURE 49 Seller Import File

```
import_text_file   Seller ()
{
  file: "seller.dat";
worksheet ()
{
  model:  "Seller";

  read Supply_Chain   SUPP = find(supply_chains, "DP Demo");

  this = SUPP.sellers;

  [ A1: name = #; ]                          // Short name for seller
  [ B1: super_seller = #; ]                  // Super seller to inherit from
  [ C1: description = #; ]                   // Long description for seller

  [ D1: forecast_horizon = "MONTHS"; ]       // Seller forecasts are in monthly buckets
}
import_record: A1 B1 C1 D1;
}
```


FIGURE 50 Seller Data File

name	*super_seller*	*description*
# Super sellers		
Market Areas		*Super seller for marketing site sellers*
# Sellers for regional forecast sites		
CEN	*Market Areas*	*Seller for Central US Market Area*
EAS	*Market Areas*	*Seller for Eastern US Market Area*
EUR	*Market Areas*	*Seller for European Market Area*
JAP	*Market Areas*	*Seller for Japanese Market Area*
MID	*Market Areas*	*Seller for Midwestern US Market Area*
NOR	*Market Areas*	*Seller for Northern US Market Area*
SOU	*Market Areas*	*Seller for Southern US Market Area*
SWS	*Market Areas*	*Seller for Southwestern US Market Area*
WST	*Market Areas*	*Seller for Western US Market Area*
# Sellers for special state forecast sites		
CA	*Market Areas*	*Seller for California Market Area*
NY	*Market Areas*	*Seller for New York Market Area*
OH	*Market Areas*	*Seller for Ohio Market Area*
TX	*Market Areas*	*Seller for Texas Market Area*

4.10.19 Seller Plan

This example illustrates an import file for reading a seller plan data file. See FIGURE 51.

FIGURE 51 Seller Plan Import File

```
import_text_file   Seller_Plan ()
{
  file: "seller_plan.dat";
worksheet ()
{
  model:  "Seller_Plan";

  read Supply_Chain   SUPP = find(supply_chains, "DP Demo");

  this = SUPP.plans;

  [ A1: name = #; ]                    // Short name for the plan
  [ B1: seller_plans.seller = #; ]     // Seller the seller plan is for
}
import_record: A1 B1;
}
```


FIGURE 52 Seller Plan Data File

plan	*seller*
Active	*CEN*
Active	*EAS*
Active	*EUR*
Active	*JAP*
Active	*MID*
Active	*NOR*
Active	*SOU*
Active	*SWS*
Active	*WST*
Active	*CA*
Active	*NY*
Active	*OH*
Active	*TX*
New WH	*CEN*
New WH	*EAS*
New WH	*EUR*
New WH	*JAP*
New WH	*MID*
New WH	*NOR*
New WH	*SOU*
New WH	*SWS*
New WH	*WST*
New WH	*CA*
New WH	*NY*
New WH	*OH*
New WH	*TX*
FY96	*CEN*
FY96	*EAS*
FY96	*EUR*
FY96	*JAP*
FY96	*MID*
FY96	*NOR*
FY96	*SOU*
FY96	*SWS*
FY96	*WST*
FY96	*CA*
FY96	*NY*
FY96	*OH*
FY96	*TX*

4.10.20 Site

This example illustrates an import file for reading a site data file. See FIGURE 53.

FIGURE 53 Site Import File

```
import_text_file   Site ()
{
  file: "site.dat";
worksheet ()
{
  model:  "Site";

  read Supply_Chain   SUPP = find(supply_chains, "DP Demo");

  this = SUPP.sites;

  [ A1: name = #; ]                    // Short name for site
  [ B1: super_site = #; ]              // Super site to which this site belongs
  [ C1: role = #; ]                    // Role that site plays in the supply chain
  [ D1: description = #; ]             // Long description for site
}
import_record: A1 B1 C1 D1;
}
```


FIGURE 54 Site Data File

name	*super_site*	*role*	*description*
# Super sites			
Manufacturers		*SUPER*	*Super site for manufacturing sites*
Distributors		*SUPER*	*Super site for distribution sites*
Market Areas		*SUPER*	*Super site for marketing sites*
# Plant sites			
BOS	*Manufacturers*	*LINK*	*Boston, MA Plant*
DAL	*Manufacturers*	*LINK*	*Dallas, TX Plant*
RIV	*Manufacturers*	*LINK*	*Riverside, CA Plant*
# Warehouse sites			
HOU	*Distributors*	*LINK*	*Houston, TX Warehouse*
LOS	*Distributors*	*LINK*	*Los Angeles, CA Warehouse*
LON	*Distributors*	*LINK*	*London, England Warehouse*
NEW	*Distributors*	*LINK*	*New York, NY Warehouse*
# Regional forecast sites			
CEN	*Market Areas*	*LINK*	*Central US Market Area*
EAS	*Market Areas*	*LINK*	*Eastern US Market Area*
EUR	*Market Areas*	*LINK*	*European Market Area*
JAP	*Market Areas*	*LINK*	*Japanese Market Area*
MID	*Market Areas*	*LINK*	*Midwestern US Market Area*
NOR	*Market Areas*	*LINK*	*Northern US Market Area*
SOU	*Market Areas*	*LINK*	*Southern US Market Area*
SWS	*Market Areas*	*LINK*	*Southwestern US Market Area*
WST	*Market Areas*	*LINK*	*Western US Market Area*
# Special state forecast sites			
CA	*Market Areas*	*LINK*	*California Market Area*
NY	*Market Areas*	*LINK*	*New York Market Area*
OH	*Market Areas*	*LINK*	*Ohio Market Area*
TX	*Market Areas*	*LINK*	*Texas Market Area*

4.10.21 Site Plan

This example illustrates an import file for reading a site plan data file. See FIGURE 55.

FIGURE 55 Site Plan Import File

```
import_text_file   Site_Plan ()
{
  file: "site_plan.dat";
worksheet ()
{
  model:  "Site_Plan";

  read Supply_Chain   SUPP = find(supply_chains, "DP Demo");

  this = SUPP.plans;

  [ A1: name = #; ]                          // Short name for the plan
  [ B1: site_plans.site = #; ]               // Site the site plan is for
  [ C1: site_plans.description = #; ]        // Long description for the site plan
}
import_record: A1 B1 C1;
}
```


FIGURE 56 Site Plan Data File

plan	*site*	*description*
Active	*CEN*	*Plan for site CEN in the Active plan*
Active	*EAS*	*Plan for site EAS in the Active plan*
Active	*EUR*	*Plan for site EUR in the Active plan*
Active	*JAP*	*Plan for site JAP in the Active plan*
Active	*MID*	*Plan for site MID in the Active plan*
Active	*NOR*	*Plan for site NOR in the Active plan*
Active	*SOU*	*Plan for site SOU in the Active plan*
Active	*SWS*	*Plan for site SWS in the Active plan*
Active	*WST*	*Plan for site WST in the Active plan*
Active	*CA*	*Plan for site CA in the Active plan*
Active	*NY*	*Plan for site NY in the Active plan*
Active	*OH*	*Plan for site OH in the Active plan*
Active	*TX*	*Plan for site TX in the Active plan*
New WH	*CEN*	*Plan for site CEN in the New WH plan*
New WH	*EAS*	*Plan for site EAS in the New WH plan*
New WH	*EUR*	*Plan for site EUR in the New WH plan*
New WH	*JAP*	*Plan for site JAP in the New WH plan*
New WH	*MID*	*Plan for site MID in the New WH plan*
New WH	*NOR*	*Plan for site NOR in the New WH plan*
New WH	*SOU*	*Plan for site SOU in the New WH plan*
New WH	*SWS*	*Plan for site SWS in the New WH plan*
New WH	*WST*	*Plan for site WST in the New WH plan*
New WH	*CA*	*Plan for site CA in the New WH plan*
New WH	*NY*	*Plan for site NY in the New WH plan*
New WH	*OH*	*Plan for site OH in the New WH plan*
New WH	*TX*	*Plan for site TX in the New WH plan*
FY96	*CEN*	*Plan for site CEN in the FY96 plan*
FY96	*EAS*	*Plan for site EAS in the FY96 plan*
FY96	*EUR*	*Plan for site EUR in the FY96 plan*
FY96	*JAP*	*Plan for site JAP in the FY96 plan*
FY96	*MID*	*Plan for site MID in the FY96 plan*
FY96	*NOR*	*Plan for site NOR in the FY96 plan*
FY96	*SOU*	*Plan for site SOU in the FY96 plan*
FY96	*SWS*	*Plan for site SWS in the FY96 plan*
FY96	*WST*	*Plan for site WST in the FY96 plan*
FY96	*CA*	*Plan for site CA in the FY96 plan*
FY96	*NY*	*Plan for site NY in the FY96 plan*
FY96	*OH*	*Plan for site OH in the FY96 plan*
FY96	*TX*	*Plan for site TX in the FY96 plan*

4.10.22 Sub-Product

This example illustrates an import file for reading a sub-product data file. See FIGURE 57.

FIGURE 57 Sub-Product Import File

```
import_text_file   Sub_Product ()
{
  file: "sub_product.dat";
worksheet ()
{
  model:  "Sub_Product";

  read Supply_Chain   SUPP = find(supply_chains, "DP Demo");

  this = SUPP.sellers;

  [ A1: name = #; ]                                    // Owning seller for Product
  [ B1: product_groups.name = #; ]                     // Product_Group for Sub_Product
  [ C1: product_groups.sub_products.product = #; ]     // Sub_Product to put into the group
  [ D1: product_groups.sub_products.std_split = #; ]   // Percent of group split to Sub_Product
}
import_record: A1 B1 C1 D1;
}
```


FIGURE 58 Sub-Product Data File

site	*group*	*sub product*	*split*
#Central US Market Area			
CEN	*STE*	*C8003001*	*50*
CEN	*STE*	*C8003002*	*50*
#Eastern US Market Area			
EAS	*STE*	*C8003001*	*60*
EAS	*STE*	*C8003002*	*40*
#European Market Area			
EUR	*STE*	*C8003001*	*50*
EUR	*STE*	*C8003002*	*50*
#Japanese Market Area			
JAP	*STE*	*C8003001*	*10*
JAP	*STE*	*C8003002*	*90*
#Midwestern US Market Area			
MID	*STE*	*C8003001*	*50*
MID	*STE*	*C8003002*	*50*
#Northern US Market Area			
NOR	*STE*	*C8003001*	*50*
NOR	*STE*	*C8003002*	*50*
#Southern US Market Area			
SOU	*STE*	*C8003001*	*25*
SOU	*STE*	*C8003002*	*75*
#Southwestern US Market Area			
SWS	*STE*	*C8003001*	*30*
SWS	*STE*	*C8003002*	*70*
#Western US Market Area			
WST	*STE*	*C8003001*	*50*
WST	*STE*	*C8003002*	*50*
#California Market Area			
CA	*STE*	*C8003001*	*10*
CA	*STE*	*C8003002*	*90*
#New York Market Area			
NY	*STE*	*C8003001*	*40*
NY	*STE*	*C8003002*	*60*
#Ohio Market Area			
OH	*STE*	*C8003001*	*50*
OH	*STE*	*C8003002*	*50*
#Texas Market Area			
TX	*STE*	*C8003001*	*30*
TX	*STE*	*C8003002*	*70*

4.10.23 Sub-Product Group

This example illustrates an import file for reading a sub-product group data file. See FIGURE 59.

FIGURE 59 Sub-Product Group Import File

```
import_text_file   Sub_Product_Group ()
{
  file: "sub_product_group.dat";
worksheet ()
{
  model:  "Sub_Product_Group";

  read Supply_Chain   SUPP = find(supply_chains, "DP Demo");

  this = SUPP.sellers;

  [ A1: name = #; ]                                          // Owning seller for Product
  [ B1: product_groups.name = #; ]                           // Product_Group for Sub_Product_Group
  [ C1: product_groups.sub_groups.product_group = #; ]       // Sub_Product_Group to put into the group
  [ D1: product_groups.sub_groups.std_split = #; ]           // Percent split to Sub_Product_Group
}
import_record: A1 B1 C1 D1;
}
```


FIGURE 60 Sub-Product Group Data File

site	*group*	*sub*	*split*
#Central US Market Area			
CEN	*CHA*	*STE*	*50*
CEN	*CHA*	*PAD*	*50*
EAS	*CHA*	*STE*	*60*
EAS	*CHA*	*PAD*	*40*
#European Market Area			
EUR	*CHA*	*STE*	*50*
EUR	*CHA*	*PAD*	*50*
#Japanese Market Area			
JAP	*CHA*	*STE*	*20*
JAP	*CHA*	*PAD*	*80*
#Midwestern US Market Area			
MID	*CHA*	*STE*	*50*
MID	*CHA*	*PAD*	*50*
#Northern US Market Area			
NOR	*CHA*	*STE*	*50*
NOR	*CHA*	*PAD*	*50*
#Southern US Market Area			
SOU	*CHA*	*STE*	*40*
SOU	*CHA*	*PAD*	*60*
#Southwestern US Market Area			
SWS	*CHA*	*STE*	*30*
SWS	*CHA*	*PAD*	*70*
#Western US Market Area			
WST	*CHA*	*STE*	*50*
WST	*CHA*	*PAD*	*50*
#California Market Area			
CA	*CHA*	*STE*	*50*
CA	*CHA*	*PAD*	*50*
#New York Market Area			
NY	*CHA*	*STE*	*60*
NY	*CHA*	*PAD*	*40*
#Ohio Market Area			
OH	*CHA*	*STE*	*50*
OH	*CHA*	*PAD*	*50*
#Texas Market Area			
TX	*CHA*	*STE*	*30*
TX	*CHA*	*PAD*	*70*

4.10.24 Supply Chain

This example illustrates an import file for reading a supply chain data file. See FIGURE 61.

FIGURE 61 Supply Chain Import file

```
import_text_file   Supply_Chain ()

{
  file: "supply_chain.dat";
worksheet ()
{
  model:  "Supply_Chain";

  this = supply_chains;

  [ A1: name = #; ]                    // Name for the supply chain
  [ B1: description = #; ]             // Long description for the supply chain
}
import_record: A1 B1;
}
```


FIGURE 62 Supply Chain Data File

name	*description*
DP Demo	*Default DP Demo Supply Chain*

Section 5 Export Files

5.1 Introduction

The *export file* specifies the contents of each data file to be written. The export file can be explained as *how to write the customer's file*. Thus, *Rhythm*® can be customized to write the data in a format that is easy for the customer to read.

An export file is a worksheet form which specifies how to export data from *Rhythm*® as ASCII data files. It can be thought of as a one row worksheet, where the row is repeated once for every line (record) of the data file. This row consists of cells. All cells in the export file correspond to a single record in the data file (a data file record is a line of tab delimited fields ending with \n). Each cell in the export file corresponds to a single field in the data file. Cells are named so that intermediate values can be saved. Export files contain a file extension of *.exp*.

☞ Note: Keep in mind that a line in a data file can create multiple models.

See the section on Import Files for related information.

5.1.1 Writing Export Files

Export files are written from the perspective of writing, not reading files. The reading part is, in general, automatically generated. When learning how to write export files, one should first use them to write a file. When that task is completed, one should then learn how this export file also describes how to read the file; all the input expressions are generated. The input expressions can be customized.

5.2 Export File Layout Format

5.2.1 Introduction

An export file can be presented with the following layout for exporting data.

5.2.2 export_file

A layout is defined as an arrangement of controls for *display* in a report. The *export_file* layout is different from standard layouts in that it does not deal with displaying anything, but deals with I/O (input / output). It allows the export file to export fields to a text file. The user declares the fields which are associated with an input record field. This is similar to specifying which cells go in the x axis on an axis-cross layout. If an export field is needed by multiple cells, then the cell can be referenced by name.

```
export_file   location()
{
     delimiter: " ";
     file: "site.dat";
     fixed_width: true;
     title_line_prefix: "#";;

worksheet (Factory  fm)
{
     [ A1: l = fm.locations; ]
     [ A2 = l.name; ]
     [ A3 = l.description; ]
}
Y: A2, A3;
}
```


5.3 Export File Format

5.3.1 Export File Format

5.3.1.1 Description

The general format for the export file follows. Characters in **bold** are typed as is. Characters within <angle brackets> are user specified values:

export_file *<export_file_name> (parameters)*
{
  file: *"<datafile_name$_1$> [, ..., <datafile_name$_n$>]"*
  delimiter: *"<char>";*
  fixed_width: *<true, false>;*
  title_line_prefix: *"<char>";*
worksheet *()*
{
  [<cell$_1$>: <output specification$_1$>;]
  .
  .
  .
  [<cell$_n$>: <output specification$_n$>;]
}
Y: *<cell$_1$ cell$_2$ . . .cell$_n$>*
}

where:

- *ouput specification* - an expression formatted as follows:
  - lvalue= {# | <expression>};

5.4 Parts of the Export File Format

Table 5 describes each part of the export file format.

Table 5: Parts of the Export File Format

Part	Description
export_file	Layout type used for export files.
export_file_name	Name of the export file (*.exp* extension).
parameters	
; (semicolon)	indicates the end of each export file statement.
file	Name of the output data file pathname(s) relative to the export file directory. Can also specify an absolute pathname. The data file name has a *.dat* extension. A layout property. When a file is written where an old file existed, the old file is renamed with the *.bak* extension The *file* property specifies which data file this export file interprets during an export operation. The export file is the data file to be written. This property can specify one or more files. *file: "<datafile_name$_1$> [, ..., <datafile_name$_n$>]"*
delimiter	Field separator that occurs in output data files. Any character may be used, including special characters. Default is the TAB character ("\t"). A layout property. *delimiter: "<char>"* Leading and trailing white space (the white space that exists between character strings and delimiter) is stripped. The following is an example of a data file which is parsed with *delimiter: ";"*. *# Comment lines begin with "#"* *# This is a data file for some supply chains* *# Name Description* *#----------* *supp_chain_1; Some supply chain* If leading and trailing whitespace was not stripped, then the description would be " Some supply chain". But the description is actually "Some supply chain".

Table 5: Parts of the Export File Format

Part	Description
before_export	The shell command to execute before writing a file. It is possible to create export files that append data to the *.dat* files instead of overwriting them by using the *before_export* and *after_export* properties of the export worksheet. By making the value of *before_export* a shell command to move the file, and the value of *after_export* shell commands to concatenate the old file with the new, the data will be appended to the *.dat* file rather than the *.dat* file being overwritten with the new information
after_export	The shell command to execute after writing a file. It is possible to create export files that append data to the *.dat* files instead of overwriting them by using the *before_export* and *after_export* properties of the export worksheet. By making the value of *before_export* a shell command to move the file, and the value of *after_export* shell commands to concatenate the old file with the new, the data will be appended to the *.dat* file rather than the *.dat* file being overwritten with the new information
fixed_width	If TRUE, make output fields of a fixed width.
title_line_prefix	The comment header for the output file.
worksheet	
output specification	The format for output specifications is as follows. Optional parts are in {curly brackets} # is the output record field converted to the data-type of the expression; it is non-nameable, i.e. cannot have something like #i. [<cell name:> lvalue= {# | <expression>};] For example: *[A1: sites.name =#;]* *[B1: sites.name = "Dallas";]*

Table 5: Parts of the Export File Format

Part	Description
Y	Specifies the output order of fields for each record / line of an export data file. This allows the user to explicitly define the order without being constrained by the order in which the lines with #'s appear in the export file. Not required. If not specified, then no fields are written. A layout property. *Y: <cell$_1$ cell$_2$ . . .cell$_n$>* For example: *[A1: name = #;]* *[B1: buffers.item = #;]* *[C1: buffers.description = #;]* . . . **Y**: *A1 B1;* Note that more fields may be defined than are used (i.e. specified by the *Y* property). C1 is defined but is not used in the example.

Section 6

Getting Started with Data and Import Files

6.1 Introduction

This section introduces you to supply chains, sites, and items, and describes how to write data files and import files for each. In this section you will find the following topics:

- Building a Supply Chain
- Adding a Site to the Supply Chain
- Adding Items to the Site

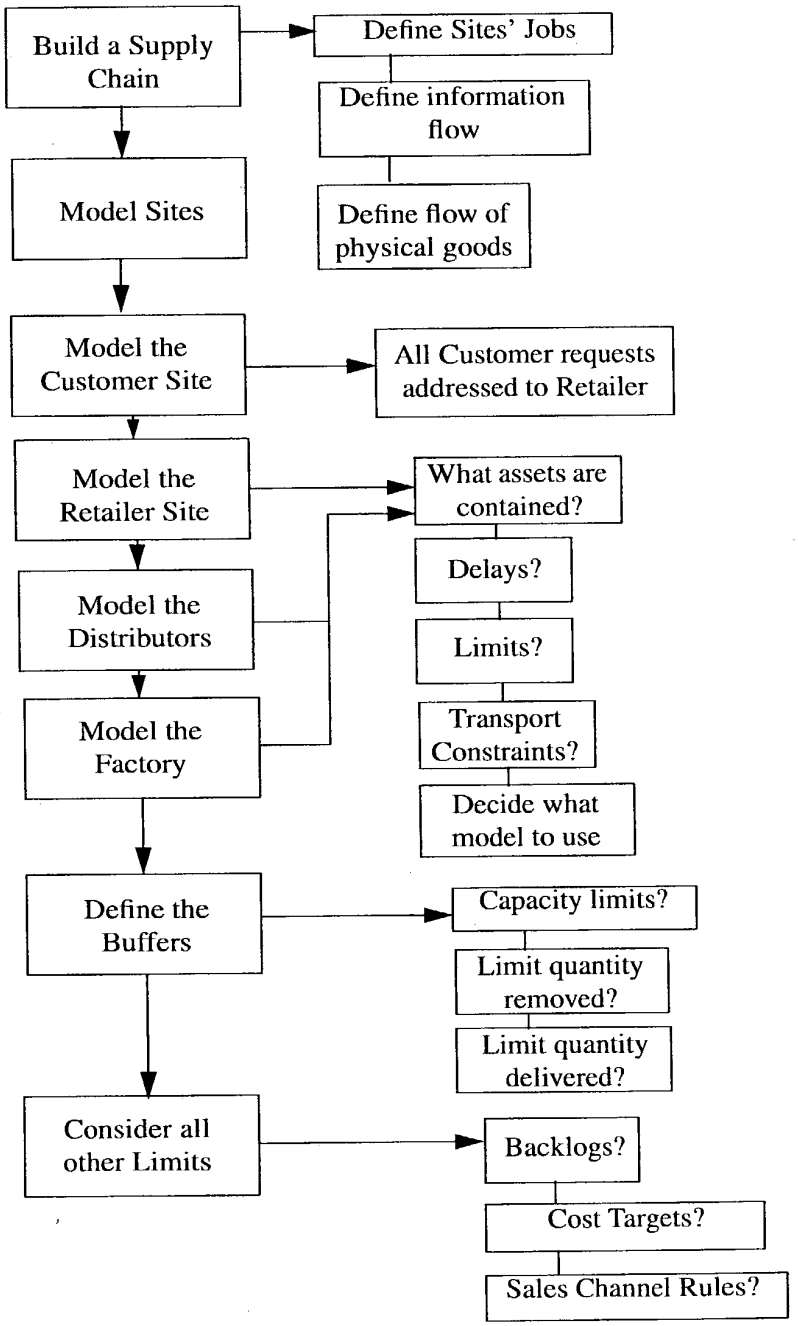
6.2 Modeling Step-by-Step

Table 6 provides a detailed explanation of the Modeling Map (See FIGURE 63). This map provides an overall view of the modeling process, of which supply chains, sites, and items as described in the following sections, form a portion. Each step of the process is discussed and examples are provided (where relevant).

Table 6: Modeling Step-by-Step

Stage	Description	Comments
1	Build a Supply Chain	In this step, define all the sites' jobs, the information flow, and the flow of physical goods.
2	Define Sites	Make sure that all sites are included.
3	Model the Customer Site	In this step, make sure that all customer requests are addressed to the retailer.
4	Model the Retailer Site	In this step, be sure to consider what assets are contained in this site, what delays may arise, what limits may arise, and what transport constraints may be present before deciding which model to use (for example, fixed_time_delay model).
5	Model the Distributors	
6	Model the Factory	
7	Define the Buffers	Consider any capacity limits, any limits on quantities to be removed, or any limits on quantities to be delivered in this step.
8	Consider all Other Limits	In this step, consider any other limits, such as backlogs, cost targets, or sales channel rules.

FIGURE 63 Modeling Map



6.3 Building a Supply Chain

6.3.1 Introduction

After completing this section, you should be able to:

- Understand what a supply chain is, and how its components are related
- Understand some basics about import files and data files
- Create a supply chain

6.3.2 Description

A supply chain consists of a set of sites, a set of plans, and a set of sellers. Sites contain static elements of the supply chain, and provide the representation of physical entities and processes. Plans are uses of the entities in the sites and identify how and when materials are produced and consumed. Sellers represent the sales force and provide information about demand. So these components tie together this way: *Sellers* provide information about product demand so that *Plans* for the *Sites* can be generated to produce the desired products.

6.3.3 Import Files

The first thing to do is create the supply chain, by writing an import file and providing a data file for it to read. An import file is a description of how to read data into models. *Rhythm*® can be customized to read the data in a format that is easy for the customer to provide.

A import file is a worksheet form which specifies how to import data to *Rhythm*® from ASCII data files. It can be thought of as a one row worksheet, where the row is repeated once for every line (record) of the data file. Cells are named, so that intermediate values can be saved. Import files contain a file extension of *.imp*.

The import file syntax (this is a subset) is essentially:

```
import_text_file   <import_file_name>()
{
    file: "<datafile_name>";
    delimiters: "\t";
worksheet ()
{
    model: "Supply_Chain";
    this = supply_chains;

    [ <cell1>: <input specification1> ]
    [ <cell2>: <input specification2> ]
    . . .
    [ <celln>: <input specificationn> ]
}
import_record: <cell1 cell2 . . .celln>;
}
```

Where:

- <import_file_name> is the name of the file without the *.imp* extension.
- <datafile_name> is the name of the file that contains the data being reading.
- <input specification> describes what to do with the value from a field in the data file.

6.3.4 Input Specifications

6.3.4.1 Description

The entire set of input specifications describes the fields in a single record of the <datafile_name> file. There is one specification per field when each field of the record is read from the file in order from left to right. In other words, each time a record of the data file is read, an entire pass is made through the input specifications (and any other statements that may follow them). In each specification, the "#" character refers to the text in the corresponding field. Fields in data files are separated (by default) with TABs.

6.3.4.2 Format

When a data file is read, models are created as necessary, and values are assigned to the fields of models. The basic format of an input specification is:

<field> = #;

where <field> is the name of a field. Many people find this counter-intuitive because it looks like the reverse of an assignment statement. This is really because what is shown is an abbreviation for something more complex (which is too detailed at this point).

6.3.4.3 Field Name

The field name may be a simple name (a single token) or a compound name (a number of tokens separated by "."). In all cases, the value of *this* assigned on an earlier line of the import file provides the context for the name. In the preceding import file example, the context is the global variable *supply_chains*. (Incidentally, *supply_chains* is a variable which contains the list of supply chain models about which Rhythm knows.) In compound names, the model hierarchy (described in the Model Reference) is traversed just like a data structure in a programming language.

Look at the description of the Supply_Chain Model in the Model Reference Manual. There are two fields: *name* and *description*. There are lists of sub-models: *sites*, *sellers*, and *plans*. There are also two List[...]'s: *top_sites* and *top_sellers*.

6.3.4.4 Setting a Field

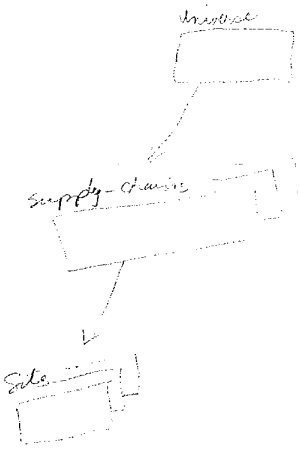
Anything that is described as a *field* (without the List[...]) is something you can set directly using the import file; it is essentially an attribute of the model. In each model, one field is designated as the key field. When you *set* the key field of a model, all existing model instances of that type are examined to see if any has a matching key. If a match is found, subsequent input specifications will apply to that found model instance. If no matching model is found, a new model instance will be created with the new key. Regardless of whether the model is found or a new one is created, as long as the same record from the data file is being processed, subsequent model references apply to the same model instance. This mechanism is referred to as *find_or_create*.

6.3.4.5 Dependencies Among Input Specifications

The previous process implies that there is a certain dependency among the input specifications in an import file. The writer of an import file does not have to worry about those dependencies, however, because Rhythm manages the dependencies automatically. Before reading the data file, the dependencies between statements in the import file are

determined and an appropriate ordering is applied to them, taking into account the dependencies.

So an input specification of *name = #;* in the import file for supply chains would read a field from the current record in the data file, and look for a supply chain which has a *name* field with a matching value, creating a new supply chain instance if a match is not found.



When a field is described as "list of ... submodels," you use the name of the list as if it were a singleton in an input specification. For example, if the global variable *supply_chains* was a list of submodels of a higher-level model called universe, supply_chains would have this entry in the Model Reference: supply_chains, a list of Supply_Chain submodels of model Universe. So when we want to talk about a specific supply chain we use the identifier of the "list of Supply_Chain submodels".

When a field is described as "List[...]" it is read-only data. In general, these are lists of things which are related to the model, but not part of it.

So an input specification which will *find_or_create* the supply chain named as a field in a data file would look like this:

name = #;

6.3.4.6 Processing a Field

This is what happens when the field is processed. The context for the tokens is the global variable *supply_chains* (because that is the value of the *this* variable). As mentioned earlier, when dealing with a "list of ... submodels", the field should be used as if it were a singleton when importing data. So the token *name* here really means *supply_chains*.name. Since *name* is the key field of a Supply_Chain model, that causes the *find_or_create* semantics. *Rhythm*® looks for a supply chain with the same name as the value of the field read from the data file, and creates it if it is not found.

To provide a description for the supply chain, use the following:

description = #;

Since each record of the data file refers to a single supply chain, once the particular supply chain is identified from its key field, all the other field references use the same supply chain. So if *name = #* created a supply chain with the name, the *description = #* will set the description for the *My supply chain* supply chain.

6.3.5 Data File Basics

6.3.5.1 Delimiters

The format of data files is more-or-less arbitrary, except that data files are record-oriented (a new-line ends each record). By default, fields are delimited with a single TAB character (delimiters = "\t";), but you can change that if you like. If you place

delimiters: ";" ;

at the beginning of your import file (near the *import* or *model* attributes), then records in the data file will be parsed using what you specify as field delimiters (in this case, a semicolon). A semicolon (;) must occur as the termination of this delimiter specification. This an example of specifying a semicolon as a delimiter.

Arbitrary whitespace is not a field delimiter. Leading and trailing white space (the white space that exists between character strings and delimiters) is stripped.

The following is an example of a data file which is parsed with *delimiters: ";"*.

```
# Comment lines begin with "#"
# This is a data file for some supply chain
# Name              Description
#-------------------------------------------------------------
supp_chain_1        Some supply chain
```

If leading and trailing whitespace was not stripped, then the description would be " Some supply chain". But the description is actually "Some supply chain".

This data file is intended to be imported by an import file which creates supply chains. It has a single data record. There is nothing special about the comment line identifying Name or Description, but it is handy because the contents of the fields need to be meaningful for the import file's input specifications.

6.3.6 Creating a Supply Chain Data File and Import File

So we have identified the supply chain for which we want to provide a site, assigned the appropriate value of *this* to be a list of submodels of that supply chain (so the *find_or_create* works as we want it to), and used the context provided by *this* to come up with expressions that will read fields from a data file and set fields in model instances as necessary. The following table shows each step in detail.

Table 7: Creating a Supply Chain Data File and Import File

Step	Action	Comments
1	Create a data file called *supply_chains.dat* which has a single data record.	This record has two fields, which use the default delimiter (TAB). • *name* of the supply chain (its key field) • *description* The *name* of the supply chain could be *Cookie Factory Chain* and the *description* can be any text you like.

```
# Comment lines begin with "#"
# Sites used by Cookie Factory supply chain
# Name                Description
#--------------------------------------------------------------
Cookie Factory        Supply chain for cookie factory
```

Table 7: Creating a Supply Chain Data File and Import File

<table>
<tr><th>Step</th><th>Action</th><th>Comments</th></tr>
<tr><td>2</td><td>Create an import file.</td><td>This is used to read the data file just created. Call this file Supply Chain.imp. Its structure is as described above. It is used to create Supply_Chain models, and import the data file we just created. The capitalization of the .dat file must match the import specification (file) in the .imp file. For example, if the .imp files specifies file: supply_Chains.dat, then the data file name should be supply_Chains.dat.

The import file could be similar to the following:

import_text_file Supply Chain()
{
file: “supply_chains.dat”;
worksheet ()
{
model: “Supply_Chain”;

this = supply_chains;

[A1: name = #;] // Get name of supply chain; create one if necessary
[B1: description = #;] // Some text about this supply chain
}
import_record: A1 B1;
}</td></tr>
<tr><td>3</td><td>Run scp_engine using the directory containing the data file and import file as the data directory.</td><td>scp_engine -data <dir name> -port 1234 &</td></tr>
<tr><td>4</td><td>Run scp_ui.</td><td>scp_ui -port 1234 &

This verifies that your supply chain was created and has a site. When you run the UI, you should end up with a main report that has something like this in it:

<table><tr><td>Supply Chain</td><td>Plans</td></tr><tr><td>Cookie Factory Chain</td><td></td></tr></table></td></tr>
</table>

6.4 Adding a Site to the Supply Chain

6.4.1 Introduction

After completing this section, you should be able to:

- Understand the different types of sites
- Add a site to the supply chain
- Do a little more with expressions in import files

6.4.2 Description

Sites are divided into three categories:

- Supplier (which supplies raw materials)
- Link (which performs some part of the activities being managed)
- Customer (which is provided with the finished goods)

In general, only one of the sites being modeled is managed. When a site is managed, Rhythm users are concerned about its inner workings, and make decisions about how it does things. Sites which are not managed are still important, but they are important as interfaces to the managed site because they produce to, or consume from, the managed site. Their inner behavior is not important.

The previous paragraph implies that there is a particular direction in which materials flow through the supply chain. This is basically true, but the flow of materials is not determined by the sites themselves, but by the operations within the managed site.

Supply chains can contain any number of sites, but at a minimum should have at least one link site (or there would be nothing interesting for Rhythm to do).

6.4.3 Import Files

6.4.3.1 Sites

Look at the description of the Site model in the Model Reference Manual. The description says that Site is a submodel of Supply_Chain. The Site's key field is *name*. When you want to write an import file that will do *find_or_create*'s on sites, you need to specify the *name* field of a field of a Supply_Chain that is described as a submodel of, or list of submodels, of type Site.

Looking at the description of Supply_Chain, we see this:

sites -- a list of Site submodels of model Supply_Chain

so we know we need to specify (somehow) the *name* field of the *sites* entity of a Supply_Chain in a way that causes a *find_or_create*. Recalling how this is done from the previous exercise, if SC is a particular supply chain, then *SC.sites* is the list of site submodels of the SC supply chain. Since this is a list of submodels, when we specify a key field of this list for import, we are actually doing a *find_or_create*. So the expression:

SC.sites.name = #;

invokes a *find_or_create* of a site with a key field value of whatever was read from the data file.

6.4.3.2 Creating a Local Variable

We can create a variable (SC) local to the import file to hold the instance of the supply chain of interest like this:

sf_const Supply_Chain SC = supply_chains.find("This One");

6.4.3.3 this

Recall that all expressions are provided with a context by the *this* variable. The value of *this* must be assigned a list. If the list is being used for *find_or_create*, the context of the expressions is the particular instance found (or created). So if we assign the *this* variable as follows:

this = SC.sites;

then the expression to *find_or_create* a particular site from a data file would be:

name = #;

which in context is actually *SC.sites.name = #;*

6.4.3.4 Finding a Supply Chain

When we want to identify a particular instance of a model, a particular supply chain for example, we use a *find* expression. *find* (a worksheet function) is applied to lists of things. So if *supply_chains* is the list of supply chains about which Rhythm knows, the expression:

supply_chains.find("This One");

finds the supply chain with a key field value of *This One*.

6.4.4 Creating a Site Data File and Import File

So we have

- identified the supply chain for which we want to provide a site
- assigned the appropriate value of *this* to be a list of submodels of that supply chain (so the *find_or_create* works as we want it to)
- used the context provided by *this* to come up with expressions that read fields from a data file and set fields in model instances as necessary.

The following table shows each step in detail.

Table 8: Creating a Site Data File and Import File

Step	Action	Comments
1	Copy the *.dat* and *.imp* files from the exercise **Building a Supply Chain** to your solution work area for this exercise.	These are the supply chain data file and import file.
2	Create a data file called *sites.dat* which has a single data record.	This record has two fields, which use the default delimiter (TAB). • *name* of the site (its key field) • *description* The *name* of the site could be *Cookie Factory* and the *description* can be any text you like.

```
# Comment lines begin with "#"
# Sites used by Cookie Factory supply chain
# Name              Description
#-------------------------------------------------------------
Cookie Factory      Supply chain for cookie factory
```

Table 8: Creating a Site Data File and Import File

<table>
<tr><th>Step</th><th>Action</th><th>Comments</th></tr>
<tr><td>3</td><td>Create an import file.</td><td>This will be used to read the data file just created. Call this file Sites.imp. Its structure will be the same as the structure of Supply_Chains.imp. There will be a couple of differences though:
• It will be dealing with a model of type Site, not Supply_Chain
• The data file it imports will be different
• A find expression will appear before the assignment of the this variable

The import file could be similar to the following:

import_text_file Sites ()
{
file: “sites.dat”;
worksheet ()
{
model: “Site”;

sf_const Supply_Chain SC = find(supply_chains, “Cookie Factory Chain”);
this = SC.sites;

[A1: name = #;] // Get the name of the site; create one if necessary
[B1: description = #;] // Some text about this site
}
import_record: A1 B1;
}</td></tr>
<tr><td>4</td><td>Run scp_engine using the directory containing the data file and import file as the data directory.</td><td>scp_engine -data <dir name> -port 1234 &</td></tr>
</table>

Table 8: Creating a Site Data File and Import File

<table>
<tr><th>Step</th><th>Action</th><th>Comments</th></tr>
<tr><td>5</td><td>Run scp_ui.</td><td>scp_ui -port 1234 &

This verifies that your supply chain was created and has a site. When you run the UI, you should end up with a main report that has something like this in it:

<table><tr><td>Supply Chain</td><td>Plans</td></tr><tr><td>Cookie Factory Chain</td><td></td></tr></table></td></tr>
<tr><td>6</td><td>Select the small icon next to Cookie Factory Chain.</td><td>This displays the Supply Chain Editor.</td></tr>
<tr><td>7</td><td>Select the tab that says Sites.</td><td>This lists the sites created for the supply chain and looks something like this.

<table><tr><td>Site</td><td>Role</td><td>Managed</td><td>Category</td><td>Delivery City . . .</td></tr><tr><td>Cookie Factory</td><td>CUSTOMER</td><td></td><td></td><td></td></tr></table></td></tr>
</table>

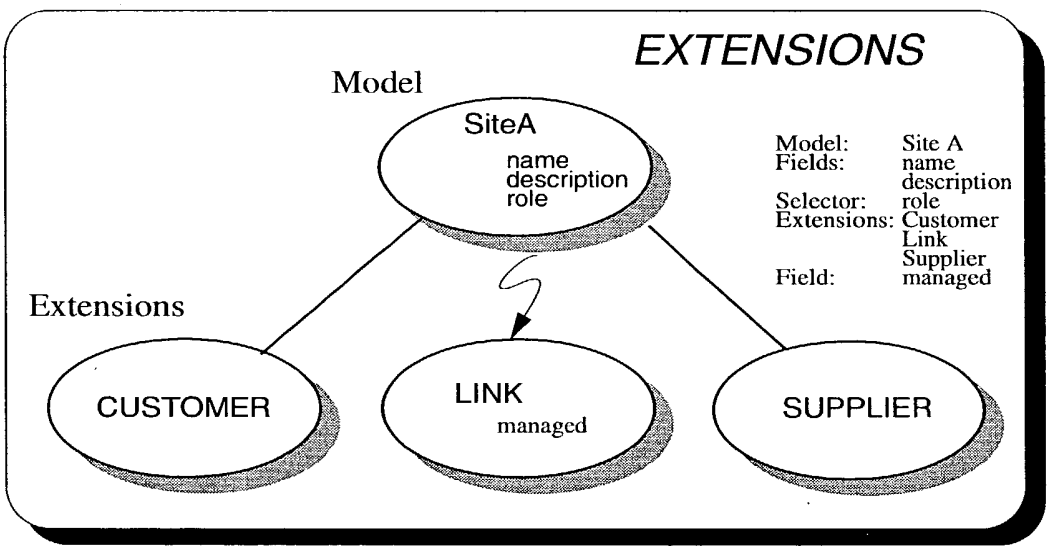
6.4.5 Changing the Role of a Site

Notice that the Role of the site is CUSTOMER. We want this to be a LINK site. To change the role of a site, we need to specify the value of the *role* extension selector. Look at the description of *role* under the Site model in the Model Reference Manual. There are three available extension selectors (See FIGURE 64):

- SUPPLIER
- LINK
- CUSTOMER

When you specify an extension selector in a data file which is read with an import file, you specify it just like it appears in the Model Reference Manual. Even though the *role* field is special, it is set just like any other field.

FIGURE 64 Extensions Example



The following table describes how to change the role of a site.

Table 9: Changing the Role of a Site

Step	Action	Comments
1	Change the *sites.dat* file to have three fields instead of two.	The first field will be the site *name*, the second will be the site's *role*, and the third will be the *description*.
2	Assign a value of LINK to the role in the *.dat* file.	The Cookie Factory site will now be a LINK site.
3	Adjust the *Sites.imp* file to read in the new field and set the site's role appropriately.	[B1: role = #;] [C1: description = #;]
4	Change the input order specification in the *Sites.imp* file.	import_record: A1 B1 C1;
5	Run the *scp_engine* and *scp_ui* to verify the results.	The site list will now look like this.

Site	Role	Managed	Category	Delivery City . . .
Cookie Factory	**LINK**	**No**		

6.4.6 Designating a Site as Managed

Now that we have a LINK site, it is important to designate whether or not it is managed. Since we want to model this site in some detail, we want it to be managed. Change the value of the second field of the *sites.dat* file to indicate that we want the site to be managed (use the value *YES*) in that field, and adjust the *Sites.imp* file accordingly:

[B1 : managed = #;]

But now we have a problem. The site's role will revert to CUSTOMER because we are no longer assigning a value to the *role* field in the import file, and a CUSTOMER site cannot be managed. When we want to assign a fixed value to a field of all the instances of a model read with a particular import file, we can specify the value directly in the import file with an OIL expression. This expression:

[D1: role = LINK;]

sets the role of all sites found / created with this import file to LINK. Note that this looks like an assignment statement, and there is no reference to #. Also note that the expression is enclosed in square brackets.

Expressions like this are usually placed after the input specifications. Add the expression to the *sites.dat* file that sets the role of all sites found / created with *Sites.imp*. Run the *scp_engine* and *scp_ui* again. This time the site list should look like this:

Site	Role	Managed	Category	Delivery City . . .
Cookie Factory	**LINK**	**Yes**		

6.5 Adding Items to the Site

6.5.1 Introduction

After completing this section, you should be able to add items to the managed site.

6.5.2 Description

All the items a site manipulates are owned by the site. It does not matter if the item is used as a raw material (consumed by the site), a finished product (produced by the site), or some intermediate stage of production. When one site provides an item to another site, there are actually two discrete items in the supply chain: the item the producing site owns, and the item the consuming site owns. Sometimes it is convenient for the two items to have the same name, but they are different items.

When an item is produced by a site, it has a delivery operation. This is the final process in the site which makes that item available for delivery to a customer. We will see more of this when we create operations.

6.5.3 Creating an Item Data File and Import File

In the previous exercise, we:

- identified the supply chain for which we want to provide a site
- assigned the appropriate value of *this* to be a list of submodels of that supply chain (so the *find_or_create* works as we want it to)
- used the context provided by *this* to come up with expressions that read fields from a data file and set fields in model instances as necessary

The following table shows how to create a data file and an import file for an item that is to be added to the site.

Table 10: Creating an Item Data File and Import File

Step	Action	Comments
1	Copy the *.dat* and *.imp* files from the exercise **Adding a Site to the Supply Chain** to your solution work area for this exercise.	These are the supply chain and site data files and import files.
2	Create a data file called *items.dat* which has data records for two items.	The records will have two fields, which use the default delimiter (TAB). Each record (line in the file) will be an item for the *Cookie Factory* site. • *name* of the item (its key field) • *description* The *names* of the items could be *Dough* and *Cookie* and the *descriptions* can be any text you like.

```
# Comment lines begin with "#"
# Items used by Cookie Factory
# Name          Description
#-----------------------------------------------------------------
Dough           The dough that gets made into cookies
Cookie          The cookie that the factory makes
```


Table 10: Creating an Item Data File and Import File

<table>
<tr><th>Step</th><th>Action</th><th>Comments</th></tr>
<tr><td>3</td><td>Create an import file.</td><td>This will be used to read the data file just created. Call this file Items.imp. Its structure will be the same as the structure of Sites.imp. There will be a couple of differences though:
• It will create Items, not Sites
• The context of the this variable will be the site Cookie Factory

The import file could be similar to the following:

import_text_file Items ()
{
file: “items.dat”;
worksheet ()
{
model: “Item”;

sf_const Supply_Chain SC = find(supply_chains, “Cookie Factory Chain”);
sf_const Site S = SC.find(sites, “Cookie Factory”);

this = S.items;

[A1: name = #;] // Item’s name
[B1: description = #;] // Some text about this item
[C1: artificial = “FALSE”;]
}
import_record: A1 B1 C1;
}</td></tr>
<tr><td>4</td><td>Run scp_engine using the directory containing the data file and import file as the data directory.</td><td>scp_engine -data <dir name> -port 1234 &</td></tr>
</table>

Table 10: Creating an Item Data File and Import File

<table>
<tr><th>Step</th><th>Action</th><th>Comments</th></tr>
<tr><td>5</td><td>Run scp_ui.</td><td>scp_ui -port 1234 &
This will verify that the items you specified have been created. When you run the UI, you should end up with a main report that has something like this in it:
<table><tr><td>Supply Chain</td><td>Plans</td></tr><tr><td>Cookie Factory Chain</td><td></td></tr></table></td></tr>
<tr><td>6</td><td>Select the small icon next to Cookie Factory Chain.</td><td>This displays the Supply Chain Editor.</td></tr>
<tr><td>7</td><td>Select the tab that says Sites.</td><td>This will list the sites created for the supply chain and look something like this.
<table><tr><td>Site</td><td>Role</td><td>Managed</td><td>Category</td><td>Delivery City . . .</td></tr><tr><td>Cookie Factory</td><td>CUSTOMER</td><td></td><td></td><td></td></tr></table></td></tr>
<tr><td>8</td><td>Select the tab that says Items.
<table><tr><td>Item</td><td>Spec</td><td>Artificial</td><td>Description</td></tr><tr><td>[unspecified]</td><td>STANDARD</td><td>Yes</td><td></td></tr><tr><td>Cookie</td><td>STANDARD</td><td>Yes</td><td>The cookie that the factory makes.</td></tr><tr><td>Dough</td><td>STANDARD</td><td>Yes</td><td>The dough that gets made into cookies</td></tr></table></td><td>This will list the items created for the Cookie Factory site and look something like this. You will notice that the two items created are artificial. That means we cannot do anything with them. None of the items we create should be artificial. Look at the description of the Item model in the Model Reference Manual. Note which field specifies the artificial attribute of an item, and modify the Items.spc file so that items created with this import file will not be artificial. Verify your solution is correct.</td></tr>
</table>

Section 7 Engine and UI Options

7.1 Introduction

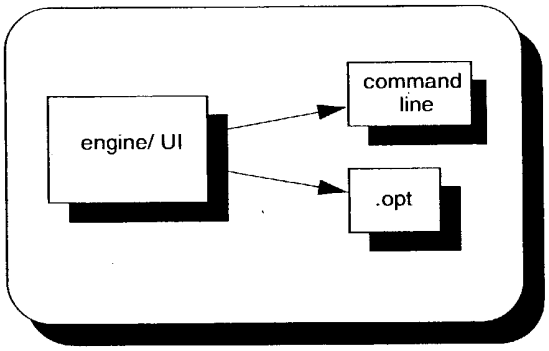
Rhythm® is designed to be easily customizable. This capability provides the flexibility desired by customers for customizing *Rhythm*® to the needs of their own manufacturing situation. User customization may be implemented through one of the following mechanisms:

- *engine - Rhythm*® options supplied at the engine level (See FIGURE 65)
- *UI - Rhythm*® user interface (UI) options supplied at the client level (See FIGURE 65)
- *reports, layouts, worksheets, styles, formats*
- *data files*

Rhythm® option *(.opt)* files allow for customization of such items as the following:

- behaviour of the engine
- behaviour of the UIs

FIGURE 65 Engine/Client Option Architecture



7.2 Overview

This section begins with a brief overview of each of the customization mechanisms. It then presents a discussion of topics which apply to both the engine and UI, and to *Rhythm*® options:

- Naming Option Files
- Specifying Options

The remaining sections discuss the following topics:

- All available *Rhythm*® engine options and UI options are listed and described
- Environment variables

7.2.1 Engine

The *engine* program runs with a specific set of options. The option values with which the program runs may be modified by supplying:

- *Rhythm*® option/value pairs on the *Rhythm*® engine command line
- *Rhythm*® option/value pairs in *.opt* files

7.2.2 UI

The *client* program runs with its own specific set of UI options. The option values with which the program runs may be modified by supplying:

- *Rhythm*® option/value pairs on the *Rhythm*® client command line
- *Rhythm*® option/value pairs in *.opt* files

7.3 Naming Option Files

The *Rhythm*® application options can be specified for either all *Rhythm*® applications or specific *Rhythm*® applications, and can be specified for an entire customer site or for particular users.

7.3.1 Naming Rules

The following rules govern the names that option files should be given:

- *Rhythm*®applications options have the suffix *.opt*.
- Options that apply to *all Rhythm*® applications should be specified in files named *i2.opt*.
- The file containing *Rhythm*® application options for the engine level should be named *scp_engine.opt*.
- The file containing *Rhythm*® options for the UI level should be named *scp_ui.opt*.
- *Site-wide* option files should be placed in the custom subdirectory of the *Rhythm*® directory. For example, if *Rhythm*® is installed in the directory */usr/local/Rhythm*, then the site-wide options would be in the directory */usr/local/Rhythm/custom*.
- *User-specific* option files should appear in the user's home directory.

7.3.2 Search Rules

The following rules explain the directories that are searched for option files. To obtain the values for options, *Rhythm*® searches a sequence of directories (called a search path) in a specific order. The value of an option found in a file in one directory overrides, or resets, the value of the same option found in a file in any of the directories in lower pri-

ority in the search path. This search sequence allows a user to override the site-wide options, which can override the i2 Technologies-supplied options:

- *Command line options* - highest level of priority. Any options specified on the command line call of a *Rhythm*® application override any of the options specified in the files described below.
- *Standard Rhythm*® *settings* - i2 Technologies-supplied defaults.
- At the same level (user-specific, site-wide, standard), *application specific* settings will override all-*Rhythm*® settings.

7.3.3 Default List of Option Files

The default list of option files to read comes from the *app_name_options* environment variable. If this variable does not (and usually will not) exist, then *Rhythm*® searches each of the option files shown in Table 11 for the *Rhythm*® options, if they exist. The following conventions are used:

- *<app_dir>* indicates the pathname to the executable file which runs the *Rhythm*® program.
- *<app_name>* indicates the *Rhythm*® application program (which may be overridden with the *name* command line option) that is being run. For example, *scp_engine*.
- $HOME indicates the user's home directory.

The table is ordered from highest (most specific; current directory) to lowest (most general; home directory) level of override priority. Command line options override option file values.

Table 11: Search Rules for Rhythm Options

Rule	File	Level	Scope
9	command line options	user specific	all *Rhythm*®
8	*<app_name>*.opt	site-wide	application specific
7	i2.opt	site-wide	all *Rhythm*®
6	*<app_directory>*/*<app_name>*.opt	standard *Rhythm*®	application specific
5	*<app_directory>*/i2.opt	standard *Rhythm*®	all *Rhythm*®; should not be altered by customers
4	*<app_directory>*/custom/*<app_name>*.opt	site-wide	application specific; may be altered by customers
3	*<app_directory>*/custom/i2.opt	site-wide	all *Rhythm*®; may be altered by customers
2	$HOME/*<app_name>*.opt	user specific	application specific

Table 11: Search Rules for Rhythm Options

Rule	File	Level	Scope
1	$HOME/i2.opt	user specific	all *Rhythm*®

7.4 Specifying Options

7.4.1 Option Types

Each option has a name and a data type. The data type is a descriptor for how the value of the option is to be entered on the command line or in the option file. The available data types are:

- *Boolean*- a Boolean flag which has a value of either TRUE or FALSE.

  As a command line option, this type may be implemented by preceding the option with an optional plus or minus sign to indicate its value:

+option	set it to a value of TRUE
-option	set it to a value of FALSE
option	toggle the TRUE/FALSE value of the option

  It may also be implemented by preceding the option with a minus sign, and entering an explicit TRUE or FALSE (case insensitive) following the option name.

  Examples:

  appname +turn_me_on -turn_me_off toggle_me
  appname -turn_me_on TRUE -turn_me_off FALSE

- *Float* - the option represents a floating point number.
- *Function* - Looks like some other data type to the user, usually *String*.
- *Integer* - the option represents an integer number.
- *String* - the option represents a character string.

7.4.2 Command Line Option

Option values may be specified on the command line. For example,

scp_engine port 6162 &

where

scp_engine	is the name of the program
port	is an integer option set to integer 6162
&	run program in the background

If an option on the command line is not recognized, a help message is displayed, and the program terminates.

7.4.3 Option File Format

The option (*.opt*) file format is modeled after X11 resource files, except that no wild-carding is performed. There is one line per option. Each line has the option name followed by a colon, then by a value. For Booleans, the value will be either TRUE or FALSE:

name: value

☞ A space may follow the colon but not precede it.

An example option file:

```
! Comments begin with !
! The following line initializes the open option with the string my_customer.i2
open:  my_customer.i2
!Turn on some command trace, so we can see what is going on
startup: supply_chains.for_each(#.plans.for_each(#.site_plans.for_each
        (#.plan_to_satisfy_all_requests)))
```


7.5 Available Options

7.5.1 Help

To view a list of the available *Rhythm*® options which can be used as command line options, the executable should be run with the *help* option. To view engine or UI options that users implement frequently, e.g. *data, host, open, port*:

scp_engine	*help*
scp_ui	*help*

To view all UI options (UI options that users implement frequently, and options that are used to customize *Rhythm*®, e.g. *default_format, default_style*):

scp_engine	*help all*
scp_ui	*help all*

To view detailed help on one particular option:

scp_engine	*help <option>*
scp_ui	*help <option>*

If an option on the command line is not recognized, a help message is displayed, and the program terminates.

7.5.2 Standard Options

Table 12 lists the engine and UI options that users implement frequently, and that are common to both the engine and UI executables (*scp_engine help* **and** *scp_ui help*). Table 13 lists the options that users implement frequently with the engine executable but that do not apply to the UI executable (*scp_engine help* only). Table 14 lists the options that users implement frequently with the UI executable but that do not apply to the engine executable (*scp_ui help* only).

Table 12: Standard Options - Engine and UI

Option	Type	Default	Definition
deadman_timer	U-Integer	FALSE	Kill engine when idle more than the indicated number of MINUTES. Zero (default) disables this option.
help	Function		Print documentation for all options. Specifies all of the known option names, types, and values.
host	String	"localhost"	Default host name. If empty, use the current machine name.
include	Function	(NULL)	Adds a directory to the list used for opening data files. If a relative pathname is given for a data file or directory, the *include* path is searched to find it. The default is "." (the current directory). *include* may be specified multiple times. Each time, the specified path is prepended to the previous path. *path* may be a single directory, or a list of directories delimited by one or more " \t," (Note: report/layout/worksheet/style/format files use the search path defined by the *user* database).
language	Function	(NULL)	Default language.
max_clients	Integer	5	Maximum number of clients allowed to connect to this engine.
max_errors	Integer	10	File reading error messages will be printed at most *max_errors* times per file.
max_records	U-Integer	1000	Maximum number of records to be viewed per table by the GUI.

Table 12: Standard Options - Engine and UI

Option	Type	Default	Definition
open	String	""	Restore data from this directory. To prevent importing on top of a restored dataset, make sure that the following entry does not exist in the *i2.opt* or *scp_engine.opt* files: *data: <directory>* Select the *Import* and *Save* (*Save As*) options in the *File* menu of the main window to restore and save the model.
option	Boolean	FALSE	If set to TRUE, report where options originate. A trace of which options come from what option files is printed.

Example of engine options reported when *option* is TRUE:

----- Reading options from ./i2.opt -----
Option **include** = *tests . /rhythm/data*
----- Reading options from ./scp_engine.opt -----
Option **open** = *electronix.i2*
Option **startup** =
supply_chains.for_each(#.plans.for_each(#.site_plans.for_each(#.plan_to_satisfy_all_requests)))
Option **command_execute** = *true*
Option **print_usage** = *mean sum realtime, mean sum memory, mean run pause*
Option **recurse_depth** = *100*
Restoring scp_engine version 3_03 A on zip started by steve at 11/14/95 15:47:13
supply_chains.for_each(#.plans.for_each(#.site_plans.for_each(#.plan_to_satisfy_all_requests)))
Handling requests from port 27111

Example of UI options reported when *option* is TRUE:

----- Reading options from ./i2.opt -----
Option **include** = *tests . /rhythm/data*
----- Reading options from ./scp_ui.opt -----
Option **splash** = *false*
Option **user** = *electronix*

option_file	Function	(NULL)	Read an options file.
port	Integer	27111	TCP Port of engine.

Table 12: Standard Options - Engine and UI

Option	Type	Default	Definition
preload	Boolean	FALSE	If set to TRUE, preload all reports, layouts, worksheets, styles, and formats. If set to FALSE (default), report files are loaded incrementally. *user <name>* then causes *scp_ui* to use the *<name>* user. In effect, the following entries are equivalent: *scp_engine -preload user <name>&; scp_ui* *scp_engine -preload&; scp_ui user <name>* The following entries are also equivalent: *scp_engine -preload user <name>* *scp_engine -preload new_user <name>* If a value of FALSE is used on the engine, then the client comes up faster because it loads the reports only when necessary.
show_progress	Boolean	TRUE	If set to TRUE, display progress messages.
startup_hook	String	"(NULL)"	Shell command to run after connecting to client or engine.
term_width	Integer	80	Terminal width in characters. Used to widen the display output when the help feature documentation is truncated. A value of 132 is usually sufficient.
version	Function	(NULL)	Display the version number and date without loading data files. Using port 0 provides the same results: *scp_engine version 3_04 A of 10/01/96* *scp_ui version 3_04 A of 10/01/96*
which_server	Boolean	TRUE	Detects another server on the same port. Set to FALSE to avoid Rhythm2 server crashes when a Rhythm3 server is using the same TCP port as a Rhythm2 server. Set to TRUE to enable the feature of reporting which *scp_engine* is using the requested TCP port. Default is TRUE because few users will be using both Rhythm2 and Rhythm3 at the same time, and because you would have to work at it to use the same port.

Table 13 lists the options that users implement frequently with the engine executable but that do not apply to the UI executable (*scp_engine help* only).

Table 13: Standard Options - Engine Only

Option	Type	Default	Definition
auto_remove_excess	Boolean	FALSE	If set to TRUE, automatically remove useless operation plans. Notice that after forecast consumption, the delivery operation for the representative_configuration.item is reduced, but the replenishment / assembly operation (if one is defined) does not change in quantity, thus creating an *excess_on_hand* in the buffer for a representative item. Those who have not used representative item might consider this scenario when reducing the quantity of a delivery operation for an item and notice that this change does not propagate upstream, i.e. to assembly / supplying operation of the buffer, thus creating an *excess_on_hand* in the buffer for the item. To have this change propagate upstream automatically, use the *auto_remove_excess* option.
data	String	"(NULL)"	Import data from this directory. To prevent importing on top of a restored dataset, make sure that the following entry does not exist in the *i2.opt* or *scp_engine.opt* files: *data: <directory>* Select *Import* and *Save* (*Save As*) options in *File* menu of the main window to restore and save the model.
dts	Integer	FALSE	If set to TRUE, save all data and options to a directory named with the dts number.

Table 13: Standard Options - Engine Only

Option	Type	Default	Definition
flow_policy_threshold	Float	1e-05	Buffer roundoff threshold value. Tells Buffers how close two numbers have to be to consider them equal. (cannot depend on perfect equality because of the limitations of computers in dealing with real numbers). For example, if a Buffer requests a supplying operation for 10 units, and it comes back with 9.99 units, that is probably OK. But if it comes back with 9.5 units, that is probably not. By using a *flow_policy_threshold* greater than 0.01, but less than 0.5, Buffer code will interpret those numbers correctly. The default *flow_policy_threshold* is 0.00001. It demands a close match by default, so as not to throw away real material thinking it is roundoff. Math processors only give a fixed range of precision. This means that the biggest Flow_Plan in or out of a Buffer needs to be no more than 160,000,000 times as large as the *flow_policy_threshold*. (That is not something that can be controlled. It is a feature of the hardware.) With the default threshold, problems may begin to occur when dealing with Flow_Plans bigger than 1,600 (= 160,000,000 * 0.00001) units. Those problems can show up in a variety of ways, but the most common is a sudden explosion of tiny Operation_Plans, each just barely above the *flow_policy_threshold*. If you need to deal with Operation_Plans bigger than 1,600 units, and you do not need to accurately represent quantities as small as 0.00001, then you will probably want to set a higher *flow_policy_threshold*, but keep in mind the following rules: • Never set it higher than 1/4 the size of the smallest units you want to represent. For example, if the smallest units in your factory are a half of something, never set it above 0.125. • Never set it to a negative number. This causes the engine to run indefinitely. • Never set it greater than 1.0.
license	String	"(NULL)"	Enter license key from the command line.
reports	String	"$I2_REPORTS"	All relative User report_directories pathnames are relative to this path.
spec	String	"(NULL)"	Read import files from this directory.

Table 13: Standard Options - Engine Only

Option	Type	Default	Definition
startup	String	"(NULL)"	Evaluate a worksheet expression at startup.
system	String	"(NULL)"	System data directory. Import data from specified directory. The default is the system directory which contains data shipped with the system. Any or all of these files can be edited and placed in another directory (see data). The files include: *measure_base.dat* - Defines default units of measure. *meta_model.dat* - Defines user defined data fields. *user.dat* - Defines the user model, which tells *scp_engine* where to load report files. See separate section titled Environment Variables for more details.
trace_resolves	Boolean	FALSE	If set to TRUE, report strategy resolve counts.
user	String	"(NULL)"	Preload all of the reports / layouts / worksheets / styles / formats for this user before servicing GUI requests. This is a name in the user *name* field in the *user.dat* file (User model).
user_data	String	"(NULL)"	User import directory.
user_spec	String	"(NULL)"	Read user import files from this directory.

Table 14 lists the options that users implement frequently with the UI executable but that do not apply to the engine executable (*scp_ui help* only).

Table 14: Standard Options - UI Only

Option	Type	Default	Definition
allow_window_growth	Boolean	FALSE	If set to TRUE, report windows are always allowed to grow to fit currently visible layouts.
batch	String	"(NULL)"	An expression to execute INSTEAD OF displaying any windows: Example 1: *scp_ui batch "echo(3+5)"* Example 2: To save the current plan to a file with a filename containing the date and time that the file is saved (now): *scp_ui batch 'save("scp_" & now & ".i2")'* The expression could be placed in a file, e.g. *file.in*, and executed either as a parameter to the engine: *scp_engine startup 'do_file("file.in")'* or as a batch client: *scp_ui batch 'do_file("file.in")'*
batch_wait	String	"(NULL)"	An expression to execute INSTEAD OF displaying any windows. Terminates after the engine command comes back with its reply. Note: use the stdout command line option so that the output goes to stdout instead of stderr (the default). The scp_ui batch option does the same as the scp_engine worksheet command. engine captures all the output generated by the command so it can be sent back to scp_ui. This does not work for the scp_ui batch option because scp_ui terminates before the reply comes back from the engine. The scp_ui batch_wait command line option works just like batch except it waits for the reply to come back from scp_engine.
display	Function		X display name.
doc_dir	String	"doc"	Directory where documentation files exist.

Table 14: Standard Options - UI Only

Option	Type	Default	Definition
initialize	String	"display_report(\"main\")"	Expression to display initial report window. *display_report* specifies the report that will be displayed when the *scp_ui* runs. An *open_report* command is sent to the engine, which evaluates the *display_report* expression. *display_report* is always compiled on the engine because it needs to read a new report.
laf	String	"windows"	Look and feel for the user interface. Value must be one of *windows* or *motif*.
max_initial_height	Integer	1000	Maximum initial height allowed for report window
maximum_initial_width	Integer	1000	Maximum initial width allowed for report window
popup_messages	Boolean	TRUE	If set to TRUE, display messages in popup window. If set to FALSE, display messages to the terminal.
resize	Function	(NULL)	One of GROW_SHIFT, GROW_FIXED, or GROW_FIXED_HORIZONTAL.
splash	Boolean	TRUE	If set to TRUE, briefly display logo window at startup.
uncomputed_height	Integer	150	Height allowed for uncomputed layout within a tab set.
uncomputed_width	Integer	450	Width allowed for uncomputed layout within a tab set.
user	String	"(NULL)"	Selects which report directories to use from the *system/user.dat* file. Pretend our user id is this user. Only works for the user who started the engine.

7.5.3 Customize Options

Table 15 lists the engine and UI options that are used to customize *Rhythm*®, and that are common to both the engine and UI executables (*scp_engine help all* **and** *scp_ui help all*). Some are only for one or the other of engine or UI but not both (and are marked as such).

Table 15: Customize Options - Engine and UI

Option	Type	Default	Definition
absolute_pathnames	Boolean	FALSE	If set to TRUE, prepend the current directory to relative pathnames. If set to TRUE, the $I2_REPORTS/ environment variable in worksheet messages is replaced by the actual pathname.
backup_prefix	String	""	When writing files, append this string to the beginning of the old file. Whenever a file is opened for write, and either this option or *backup_suffix* (or both) is a non-empty string, any existing file is renamed before the open smashes it.
backup_suffix	String	"~"	When writing files, append this string to the end of the old file. Whenever a file is opened for write, and either this option or *backup_prefix* (or both) is a non-empty string, any existing file is renamed before the open smashes it.
boolean_false	String	"0FfNn-"	The set of characters which convert to "FALSE". The first character is used for output.
boolean_format	Function	False, True	Default format for Boolean values.
boolean_true	String	"1TtYy+"	The set of characters which convert to "TRUE". The first character is used for output.
buffer_size	U-Integer	8192	ASCII file buffer size.
char_format	Function	%c	Default format for characters.
default_col_title_style	String	"Column_Title"	Style to use for axis cross column titles.
default_format	String	"default"	The default format to use for all controls.
default_row_title_style	String	"Row_Title"	Style to use for axis cross row titles.
default_style	String	"Default"	Style to use for attributes which nothing else specifies.
default_value_cell_style	String	"Value_Cell"	Style to use for axis cross value cells.
deleted_error_display	String	"<DEL>"	What to display in a worksheet for the DELETED value.

Table 15: Customize Options - Engine and UI

Option	Type	Default	Definition
delimiters	String	"\t"	Default field separator character set.
dts_directory (engine only)	String	"/tips/data/dts/"	Directory where DTS data is saved.
editable_style	String	"Editable"	Default style to use for cells where editable_p == TRUE.
error_edit_style	String	"Error_Edit"	Style to use for cells which contain editing errors.
exception_style	String	"Exception"	Style to use for cells which contain errors.
file_type	String	"(NULL)"	Default datafile extension (e.g. <dir>/<file>.<file_type>). Example: dat
focus_style	String	"Focus"	Style to use for cells which have the keyboard focus.
font_increment (UI only)	Integer	0	Integer by which to increase / decrease all font sizes.
galaxy (UI only)	Function		Arbitrary Galaxy options.
hex16_format	Function	0x%04x	Default format for 2 byte hexadecimal numbers.
hex32_format	Function	0x%08x	Default format for 4 byte hexadecimal numbers.
hex8_format	Function	0x%02x	Default format for 1 byte hexadecimal numbers.
hex_format	Function	0x%x	Default format for hexadecimal numbers.
hidden_style	String	"Hidden"	Style to use for cells which contain hidden information.
int_format	Function	%d	Default format for integers.
new_user	String	"New_User"	All new users inherit from this user. The default New_User gets its directories from the [unspecified] user. When *scp_ui* runs, and there is no entry in the user database for that user, the path in *new_user* is copied. *new_user* can be useful when all users are to be started with a set of default reports See separate section titled User for more details.
nonexistent_error_display	String	""	What to display in a worksheet for the NONEXISTENT value.
ptr_format	Function	0x%08x	Default format for pointers.
recurse_depth	Integer	1000	Maximum recursion depth for the 'recurse' function.
recurse_items	Integer	10000	Maximum number of items computed in a recursive function before it truncates.

Table 15: Customize Options - Engine and UI

Option	Type	Default	Definition
reference_error_display	String	"<REF>"	What to display in a worksheet when using a cell value which has an error.
sced	Function	TRUE	seed for rand() functions. If set to TRUE, this option causes the random number generator to produce a consistent pattern each time it is run. The random number generator is used in making arbitrary choices.
selected_style	String	"Selected"	Style to use for cells which are selected.
specfile_type	String	"spc"	File type (extension) for specfiles.
strict_conversion	Boolean	FALSE	If set to FALSE, string->number conversions can have garbage at the end of the string.
tab_width (UI only)	Integer	8	Default tab width, in characters.
timezone	Function	(NULL)	Set the Local Timezone. 0 is UTC (Greenwich time), 6 is CDT (Central Daylight). This option will more often be applicable to NT systems than UNIX systems because NT systems are sometimes not configured for the correct timezone. The default comes from the environment variable I2_TIMEZONE. If this variable is not defined, then the default comes from the system.
tips_dts_save (engine only)	String	./bin/tips_dts_save	Script to use for the dts option.
uncomputed_error_display	String	"<>"	What to display in a worksheet for cells which are not computed.
value_error_display	String	"<VAL>"	What to display in a worksheet for internals errors (overflow, etc.).
worksheet_error_display	String	"<ERR>"	What to display in a worksheet for worksheet formula errors.

7.6 User

7.6.1 Usage

The *User* data file (*system/user.dat*), and the *new_user* and *user* command line options determine the set of worksheets, layouts, and reports that are read by *Rhythm*® at start-up:

- *new_user* - all users get the contents of the directory specified in records (in the user.dat file) that have the user name value specified for *new_user* copied into their report directories.
- *user* - selects which report directories to use from the *system/user.dat* file.

Suppose the following command line is used, and the *user.dat* file contains the data shown in FIGURE 66:

scp_ui user demo

For all records in the *user.dat* file that have a user *name* of *demo*, all reports that exist in the *directory* specified in those records are read (these directories appear in the *reports* directory in the directory from which the *scp_engine* and *scp_ui* are run). In this case, the directories that would be read are:

- *reports/basic*
- *reports/calendar*
- *reports/demo*
- *reports/rhythm*
- *reports/oil*
- *reports/mpps*
- *reports/rhythmlink*

FIGURE 66 User Data File

name	*full_name*	*organization*	*directory*	*user*	*Sequence*
[unspecified];	*Default;*	*;*	*basic;*	*;*	*1001*
[unspecified];	*Default;*	*;*	*calendar;*	*;*	*1002*
[unspecified];	*Default;*	*;*	*rhythm;*	*;*	*1004*
[unspecified];	*Default;*	*;*	*oil;*	*;*	*1005*
[unspecified];	*Default;*	*;*	*mpps;*	*;*	*1006*
[unspecified];	*Default;*	*;*	*rhythmlink;*	*;*	*1007*
New_User;	*New User;*	*[unspecified];*	*;*	*[unspecified]*	
# To run with the i2 demo reports, execute: scp_ui user i2					
i2;	*i2 demo;*	*;*	*$I2_HOME/tests/reports*		
i2;	*i2 demo;*	*;*	*oil*		

```
# To run with the dp demo rhythm reports execute:   scp_ui   user  demo
demo;          Demo;       ;              demo;
demo;          Demo;       ;              ;          New_User
```

All unknown users get the *contents* of user *New_User* copied into them. The default *New_User* gets its directories from the *[unspecified]* user. So, if the following command line is used by the user who started the engine (super-user):

scp_ui

Then since no user was specified on the command line, the default user name of *New_User* is used. For all records in the *user.dat* file that have a user *name* of *New_User*, all reports that exist in the *directory* (in the *reports* directory in the directory from which the *scp_engine* and *scp_ui* are run) specified in those records are read. In this case, the directories that would be read are:

- *reports/basic*
- *reports/calendar*
- *reports/rhythm*
- *reports/oil*
- *reports/mpps*
- *reports/rhythmlink*

7.6.2 Connecting Multiple UIs to an Engine

To authorize a client to connect to your machine, you can give him / her permission using the system/user.dat file. An example of that is when your loginid on UNIX (where you run the engine) is steve whereas your loginid on NT (where you run the client) is "Steve Chaples" or "steve chaples". You would modify the system/user.dat and copy the New_User line as follows:

New_User;	*New User;*	*[unspecified];*	*;*	*[unspecified]*
Steve Chaples;	*New User;*	*[unspecified];*	*;*	*[unspecified]*
steve chaples;	*New User;*	*[unspecified];*	*;*	*[unspecified]*

7.6.3 Security

All users of the UI must be registered in the *system/user.dat* file except the *super-user* (the user who started the engine). The *super-user* can run as anyone via the *user* command line option.

If a user's name does not appear in *system/user.dat*, then he is not given access to the engine. This is a security feature, as illustrated by the following example.

Suppose a company is running two engines. Some users need to be restricted from accessing one of those two engines. This restriction is accomplished by forcing all users to be registered in *user.dat*. The only exception is the user who starts the engine. That user must have write permission to *system/user.dat* (or the save/restore file) in order to add new users.

7.7 Adding User Defined Fields

7.7.1 Description

Models (those with a 'plist' field) can be extended with additional user defined fields. The *meta_model.dat* file adds additional fields to various models. Each row in the file represents a different model. Each column represents a field in each model. Each field has a name, value-type, and description. The implementation of the field stores the field data in the model's *plist*, which is essentially a vector of name/value pairs.

Model	*Value_Type*	*Field_Name*	*Description*
Data;	*Computed_String;*	*description;*	
Buffer;	*Symbol;*	*short_name;*	*Buffer category*
Item;	*Symbol;*	*short_name;*	*Buffer category*
Product;	*Symbol;*	*short_name;*	*Buffer category*
Product_Group;	*Symbol;*	*short_name;*	*Buffer category*

7.7.2 Lists of User Defined Fields

Often, Object Interaction Language (OIL) lists are owned by some piece of Rhythm. This saves Rhythm from always copying lists by value, which would be expensive. If you were to modify a list, it could have undesirable side-effects in the rest of Rhythm. so that is not allowed.

Suppose you have the following entry in the *system/meta_model.dat* file:

Buffer; List[Number]; oh; Buffer oh numbers

Rhythm does not allow you to modify parts of a list, so the following does not work:

[C1: buffer.oh[1] = A2;] (not allowed)

Rhythm does allow you to assign entire lists, so the following works:

[C1: buffer.oh = list(buffer.oh, A2);]

7.8 Adding UI Only Fields

UI only fields can be added to most models. To add a UI only field to a model, specify the field being added and its type for the required model inthe .dat file. Once added, the field can be used just as all other fields in the model are used.

7.9 Environment Variables

7.9.1 Environment Variables for the Engine

Rhythm® sets the following environment variables for use by the *system* command in the engine:

- I2_APP - application name (either *scp_engine* or *scp_ui*)
- I2_DATA - from the *data* option in the engine
- I2_HOME - directory where the executable exists
- I2_TIMEZONE - provides the default value for the *timezone* command line option for the *scp_engine* and *scp_ui*.
- I2_PID - process ID
- I2_PORT - TCP port number
- I2_REPORTS - $I2_HOME/reports

7.9.2 Environment Variables and Pathnames

Rhythm® does environment variable substitution within pathnames, when the environment variable is at the beginning of the pathname. The following environment variables get default values:

- I2_HOME - the directory where the executable exists
- I2_DATA - I2_HOME/data
- I2_REPORTS - I2_HOME/reports

These environment variables are used when setting up *user.report_directories* or the *include* option to make it easy to specify files relative to the executable, and to make it easy to switch to a different set of data or reports.

Relative pathnames in *user.report_directories* get the contents of the *reports* option prepended automatically. The default value of the *reports* option is $I2_REPORTS.

7.9.3 Printing

The *Print Report* option in the *File* menu prints a copy of the layout for this report to the printer.The *Print Report* option runs the following action (see the *application.rpt* file):

```
action:  print_layout =
             do(setenv("I2_PRINTFILE", "/tmp/print." & getenv("I2_PID")),
print_layout(getenv("I2_PRINTFILE")),
system(getenv("I2_PRINT") ? "lpr $I2_PRINTFILE"),
system("rm -f $I2_PRINTFILE"));
```

This saves the layout (not the report) that has input focus. (Be sure to select the layout to be printed before using the *Print Report* option, so that the layout has input focus.) The default is to use lpr to do the printing. If the host uses something other than lpr, then the I2_PRINT environment variable needs to be customized. The value of I2_PRINT can be

any shell command. One of the easiest ways to set I2_PRINT is in the *scp_ui.opt* file. For example,

```
initialize: do(display_report("main"),
            setenv("I2_PRINT", "a2ps -F5.75 -ns -f $I2_PRINTFILE | lpr -h"))
```

Note that someplace in the shell command, a reference to $I2_PRINTFILE, which is the temporary file that contains the ASCII text being printed, should be made.

Section 8 Topics

8.1 Introduction

This section describes the key topics that are addressed by *Rhythm*®.

8.2 Assigning Priority to Demand (Item Requests)

Priority can be assigned to demand (item requests) as necessary, for example to realize preferred treatment of A customers. This is done by defining several buffers for each end-item and establishing stealing operations as alternates for A but not for B customers. This works only in make-to-stock environments

8.3 Available To Promise (ATP) and Automated Order Promising

8.3.1 Description

This section defines and describes Available To Promise (ATP) as used in Rhythm. It lists and briefly describes some of the models that Rhythm uses when determining ATP and performing order promising processes. It also lists and briefly describes the user interface windows that provide ATP information, and that you use to perform order promising.

8.3.2 Available To Promise

ATP is an actual quantity of inventory, and it also refers to an order promising technique used to establish realistic delivery dates for customer orders.

Rhythm differentiates between actual and forecast ATP quantities. As a *forecast* quantity, ATP is the seller allocation less consumed quantity. As an *actual* quantity ATP is inventory plus statements of supply (such as manufacturing orders and scheduled receipts) less customer orders (due and past due). When working with forecasts, Rhythm provides information about the current status of forecasts, including consumed and ATP quantities. When working with requests, Rhythm provides information about ATP quantities and dates that may be used to satisfy requests. Request is term used in Rhythm that can refer to forecast, a customer order, or particular items on an order.

As an order promising technique, ATP involves the use of a number of models that allow customizing how the ATP quantity is determined. Some of the Rhythm models that contain information used to determine ATP quantities are listed below.

- *Item_Request* - models a request for a supply of a quantity of a particular item.
- *Delivery_Request* - models a request for a set of items to be delivered together.
- *Request* - models a set of delivery requests that are issued together and a promise is to be offered for the whole
- *Item_Promise* - models an agreement to supply/consume a quantity of a particular item
- *Delivery_Promise* - models a supplier's response to a customer's delivery request.
- Promise - models a supplier's response to a request. A promise exists for every request.
- *Available_To_Promise* - models a forecast promise. This quantity has been allocated for use by the seller, but has not yet been consumed by an actual request (order). This is the amount available for consumption to form a promise for the item request.
- *Seller_Plan* - models a seller's master plan. It includes the seller's forecast, committed sales, allocations, and promises.
- *Product* - models how an item request is matched up with a product
- *Forecast* - models the forecasts and allocations (the master plans) for a product or product group of the seller or the seller organization.

Some fields of the ATP model include the following:

- product - for which there is ATP
- forecast - for product from which ATP is computed
- forecast entry - from which ATP is computed
- dates - which can be promised with this ATP
- quantity - the quantity of ATP product

Detailed information about these and other models is provided in the *Rhythm* Model Reference Manual.

Rhythm UI windows that provide information for ATP and order promising are the following:

- *Request*
- *Delivery Request*
- *Item Request*
- *Item Promise*
- *Forecast*
- *Forecast Entry*

Each of these windows is described in the *Rhythm* Standard Reports Manual.

8.3.3 Order Promising

Order promising in Rhythm is part of the larger process of demand management. To see where order promising fits in, it's helpful to have an overview of demand management. The items listed next describe the pieces of demand management.

- Forecasting and forecast management - estimating the sales potential for each product, independently.
- Master planning - determining how to allocate capacity and materials to best meet the forecasts for the products. Master planning determines how much of each product is available and when.
- ATP calculation - determining quantities based on the allocations received for each forecast request. The allocation quantity less consumed quantity becomes the ATP quantity.

See the Demand Management and Request and Promise sections in this manual for more information about order promising.

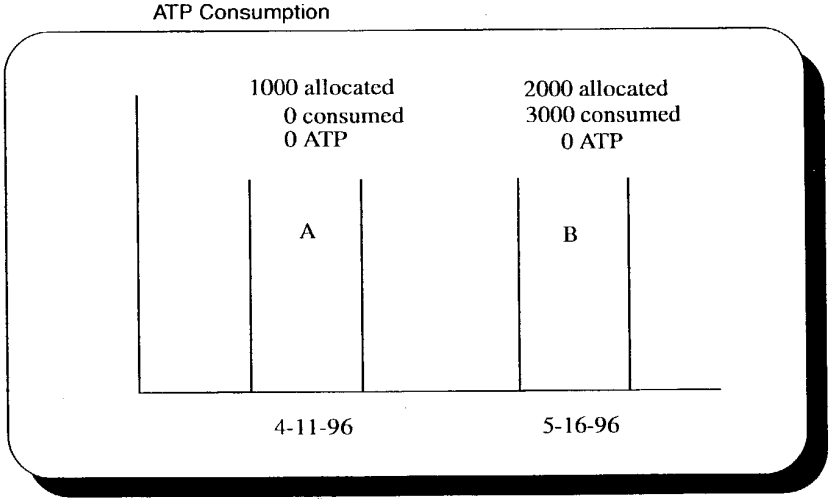
A customer may request a specific item on a specific day. The request may be satisfied by any one of a number of products. Each product has independent availability and pricing. The item request may be satisfied from allocations belonging to the seller's organization, and not necessarily just to the seller making the request. Rhythm presents a list of all the options for meeting the customer's request. The system provides the quantity that can be delivered by the requested date. If the requested quantity cannot be provided by the requested date, Rhythm also gives the date by which the total requested quantity can be delivered. Rhythm users help the customer choose the most satisfactory option, based on price and timing. The information is presented in the Request Editor report, on

the Request and Quote layout. See the *Rhythm* Standard Reports manual for information about the Request Editor.

8.3.4 ATP Consumption

It is possible to see a situation in which Rhythm shows an allocated quantity, but zero consumed and zero ATP. This occurs because quantity information is displayed in time buckets, and the quantity is consumed in a different, later time bucket than the one in which is it allocated. See FIGURE 67. There appears to be a discrepancy in time bucket A, where a quantity of 1000 is allocated, but there is zero consumed and zero ATP. What has happened is that the 1000 is consumed in time bucket B. The information is displayed for what happened in a particular time bucket. The bucket in which the order is delivered (bucket B) gets its consumed quantity increased by the entire amount of the customer order, but it may take ATP from some previous buckets if needed.

FIGURE 67 ATP Consumption



Order Quantity: 3000

8.4 Batching

8.4.1 Description

A batch is defined as a group of orders that are processed simultaneously on a batch resource. Typical types of batches and what defines them as batches are volumes of paint (container size), bunch of cookies (capacity of a cookie tray or oven), capped bottles (size of capping carousel).

A batch capacity is defined as the amount that must be run together on the machine at one time. It is typically constrained by a minimum and a maximum.

Batching affects the PST's for the orders in the batch and the runtime for those orders.

8.4.2 Purpose

Batching models resources that process groups of orders simultaneously, instead of sequentially.

8.4.3 Procedure

To perform batching of orders at selected resources, the SHARED_USE load_policy extension of the resource model should be specified. The FIXED_QUANTITY size extension of the resource model should then be selected.

A FIXED_QUANTITY size resource has a single fixed size limit at all times. The size of an operation is measured by converting the units of the operation plan to the unit of measure of the *max_size*. Thus, if the *max_size* is 500 kg, the size of an operation plan will be the units of the operation plan converted to kg. If the *max_size* is unitless, then the number of units of the operation plan is used.The default value for *max_size* is the common case of *oo* (infinite) which means there are no size limits.

8.4.4 Example

An operation that processes batches of a fixed quantity (an oven) is handled as follows. If the oven allows jobs to be taken in and out at any time, such that these activities occur:

add job1 at 1:20
add job2 at 1:45
add job3 at 1:50
remove job2 at 1:55
. . .

Then:

- specify the SHARED_USE load_policy extension of the resource model
- specify the FIXED_QUANTITY size extension of the resource model

If the oven does not allow ins and outs at any time, this is not supported.

8.5 Calendars

8.5.1 Description

Rhythm calendars allow you to model patterns for specified days, weeks, months, and years. Calendars are used with resources, buffers, and yields. A calendar can be tied to a resource to model changes in efficiency over time. For example, new equipment or processes may be phased in over a period of time, and efficiency increases until the ramp-up is complete. A calendar can model the change in efficiency over specified periods of time. You can also attach calendars to operators, defining different efficiencies for different operators. With buffers, you can use calendars to model things such as a regular pattern in supplies to a buffer. For example, if 1,000 items are received in a raw material buffer once every two weeks, that quantity can be modeled in a calendar with a time pattern of once every two weeks. After defining the calendar, it is then attached to the buffer or resource.

Special calendars called subcalendars can be used to model other kinds of regular patterns, such as holiday and maintenance schedules. The subcalendars are then used as part of a calendar to develop a complete model without requiring a lot of duplicate effort.

8.5.2 What Can You Use Calendars to Model?

You can use calendars to model things such as changes in efficiency over time, or regular changes in buffer quantities. Each calendar has an entry value, and the type of entry value defines what the calendar models for the specified dates and times. Some of the things you can model and the entry values you use are as follows:

- Efficiency - A calendar can model variances in resource efficiency over time. As described above, efficiencies may vary with different operators, or when new equipment is implemented. Specify the NUMBER entry value, and enter a number that is a percentage.
- Capacity - A calendar can model the capacity of a resource. Specify the QUANTITY entry value, and enter the capacity quantity for the specified dates and times.
- Supplies to buffers - A calendar can model regular deliveries to a buffer. Specify the QUANTITY entry value, and enter a quantity that is the amount delivered to the buffer on the specified dates.
- Setups allowed - A calendar can model that only specified equipment set-ups are allowed at defined dates and times. Use the SYMBOL entry value and enter a symbol that refers to a specific set-up or class of set-ups.
- Task time - A calendar can model the time required to complete a specified task. For example, you may define the time required for warm-up or cool down of equipment, and the time may vary by season. Specify the TIME entry value, and enter time, such as 2.5 hours.

8.5.3 Creating a Calendar

When creating a calendar you can copy an existing calendar that is similar to the one you want to create, or you can create a completely new calendar. If you copy an existing calendar you then make changes to the copy to develop the new calendar.

8.5.3.1 Entry Value

The first thing you need to determine is the calendar entry value. The calendar entry value specifies which type of value this calendar models. The possible entry values and how they are used are as follows:

- NUMBER - a percentage that is used to model efficiencies. For example, efficiencies may vary on different shifts, and this can be defined by specifying 100% for one shift and possibly 80% for another. This can also be used to model a change in efficiency when new equipment is introduced. You might specify 50% initially for a specified time period, and change the percentage over time until it reaches 100%.
- QUANTITY - a specific quantity, used to model changes in things such as supplies to a buffer. If a raw material buffer receives a supply of 5,000 every two weeks, you can specify that in a calendar using QUANTITY.
- NUMBER_QUANTITY - used when a calendar needs to model numbers and quantities.
- SYMBOL - a symbol that may represent something such as a set-up. For example, this entry is used if you wish to model a situation in which only certain set-ups are allowed during specified periods of times and dates.
- TIME - a time that is used to model a situation such as a fixed amount of time for an operation, but the time may vary over time. For example, the amount of time required for warm-up of a resource may be shorter during the summer than during the winter, because the equipment is at a different temperature before warm-up. You can define a calendar with a TIME entry, and specify one value for warm-up during the summer months and a different value for the winter months.

The calendar entry value is specified in the *Calendar Editor* for each calendar.

8.5.3.2 Calendar Entry

Calendar entries describe the features of a particular item in a calendar. A calendar entry includes things such at the dates and times for which the entry is effective, the rank of the entry (used when entries overlap), and the day pattern.

A calendar may have only one entry, or it may have may have many entries. For example, a calendar that models efficiencies may have separate entries for each shift in a typical 24-hour day, with different efficiencies defined for each shift.

The information defined in a calendar entry includes the following:

- *Calendar Entry* - the calendar entry name that appears on the *Calendar Editor*
- *Calendar* - the name of the for which the entry is defined
- *Name* - the name of the calendar entry
- *Description* - a description of the calendar entry
- *Effective* - the dates during which the entry is effective

- *Daily Start* - the starting time of day for the entry
- *Daily End* - the ending time of day for the entry
- *Rank* - the rank of this entry, used when entries overlap. The highest rank takes priority when ranks overlap. A rank of -INFINITE means that this is the lowest possible rank. When specifying rank, allow for later adding entries that may rank between existing entries.
- *Charge* - the charge associated with the entry, used for entries such as overtime
- *Value* - the type of value of the entry, such as Number or Quantity.
- *Number/Quantity* - this is the actual value related to the above value item. It is titled Number, Quantity, or another value type as selected.
- *Day Pattern* - the day pattern for the entry, such as weekdays, weekends, or yearly.

8.5.3.3 Subcalendars

Subcalendars are separate calendars that have modeling information used by a top level calendar. In a subcalendar you can specify calendar information that is used by several top level calendars, and avoid the duplicate effort of entering this information a number of times. These calendars can specify information about things such as holidays. Subcalendars simplify the task of modeling events that effect normal calendar patterns.

8.5.4 Reading Calendar Data

To read calendar data, use expressions such as the following:

```
[ D1: " "; format:dd_mm_yy; ]
[ E1: " "; format:dd_mm_yy; ]
```

This will provide dates that are parsed according to the format specified. Then do the following:

```
entries.effective = date_range(D1, E1);
```

8.5.5 More Information

See the following topics in the *Supply Chain Planner Standard Reports Manual* for more information about defining calendars and for descriptions of the calendar editors.

- *Calendar*
- *Calendar Entry*
- *Subcalendar*

8.6 Date Effectivity via Families

Date effectivity specifies a date range in which some entity is valid. Thus, Process10 might be valid in May, while Process11 valid in June. "valid" gets a tight definition, such as "first operation in the process must start and end in the date effective period". That definition is enforced by a problem detector, so a problem arises if you plan the Process10 operation outside of May.

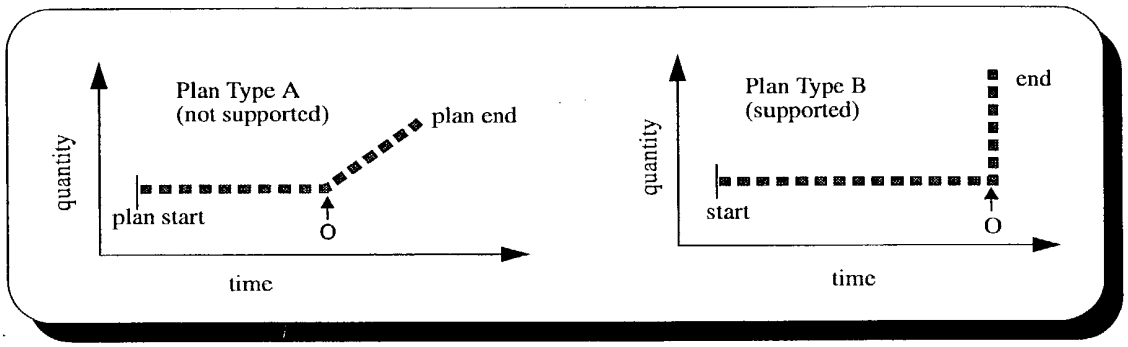
Families can be used to simplify setting up the data. If Process11 is a slight variation on Process10, they both can be siblings of a higher level process which is never date effective. Or, Process10 could be the ancestor, and Process11 its descendent.

Given that, one of the resolving techniques for date effective problems is to switch processes.

8.7 Delivery Plan Due Date

A delivery request or delivery promise has a due date which is a date range. This means that requested items need to be supplied between the start and the end dates. There are certain flows in the operation that would supply items over a period of time. Typically, the supplying pattern of an operation plan would look as in FIGURE 68.

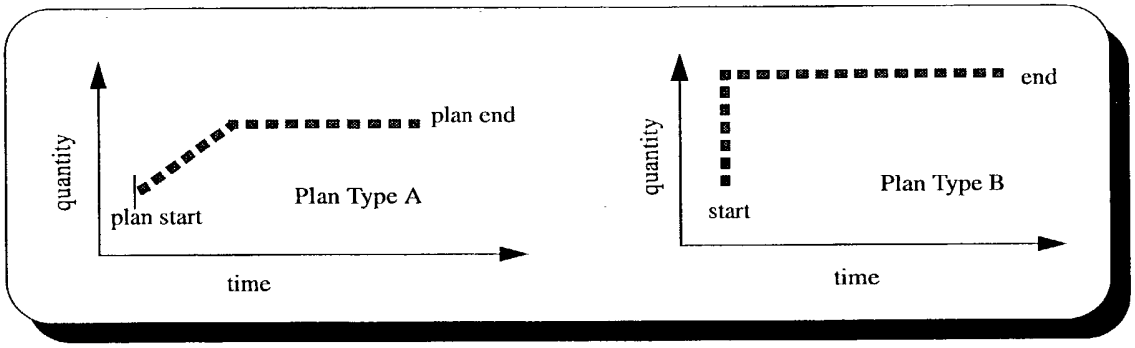
FIGURE 68 Operation Plan Supplying Pattern



A delivery actually begins at the time indicated by O. To determine whether a delivery is late or early, the date range from plan O to plan end should be compared against the date range from the request start time to the request end time. In addition there is the *rate_start* field in the Request model which determines the shape of the supply curve of the item quantity. So, in addition to fitting the plan between a date range, the shape of the supply curve needs to be controlled. Restrictions are needed in the operation plan. Currently no functions exist to find O (the time at which a delivery actually occurs) or to move the plan using restrictions on O (or for shape control), so only plans of type B are supported. The plan end time should occur between the request start time and the request end time.

FIGURE 68 is with respect to a buffer. If the operation has a deliver motive, it has a consuming flow with respect to the buffer. With respect to the request, this flow is supplying to the request. The picture would look as in FIGURE 69:

FIGURE 69 Delivery Plan Due Date



8.8 Demand Management

8.8.1 Description

Demand management (or order management) involves the recognizing and managing of all demands for products to ensure that the master scheduler is aware of them. Demand management includes the following functions:

- Forecasting
- Order requests
- Order promising
- Warehouse requirements
- Supply chain orders
- Parts requirements

In Rhythm, the management of demand involves the following concepts:

- Sellers define and manage products. Sellers model the responsibility for forecasting demand, committing to sales, and managing allocations.
- Plans - model the current and future activities of a supply chain. Seller plans model the seller's actions.
- Products are forecasted.
- Request and Promise is the mechanism by which different authority domains talk to each other. The Request and Promise mechanism includes the following functions:
  - turns forecasts into available-to-promise
  - implements actual demand and its fulfillment
  - handles communication between Sites in separate authority domains

8.8.2 Relevant Models

The following models work together to implement what is traditionally called demand management. Simplified variations of these models are often called orders.

- Request
- Delivery Request
- Item Request
- Promise
- Delivery Promise
- Item Promise

8.8.3 Product Modeling

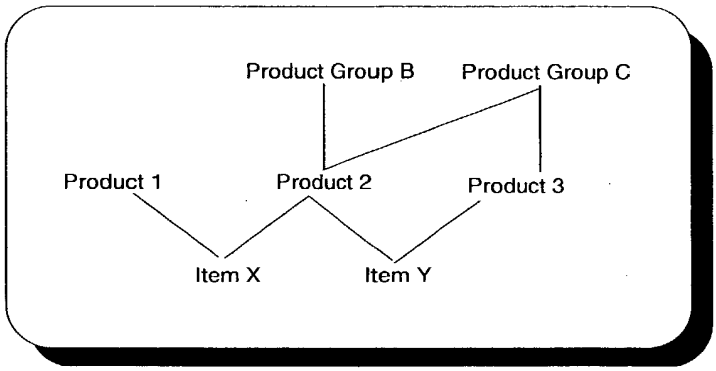
Sellers each have a collection of products that they manage. A product is one or more items:

- Which are available to a set of customers
- With a certain delivery lead time

- At a certain price

FIGURE 70 shows the relationships between product groups, products, and items.

FIGURE 70 Product Modeling

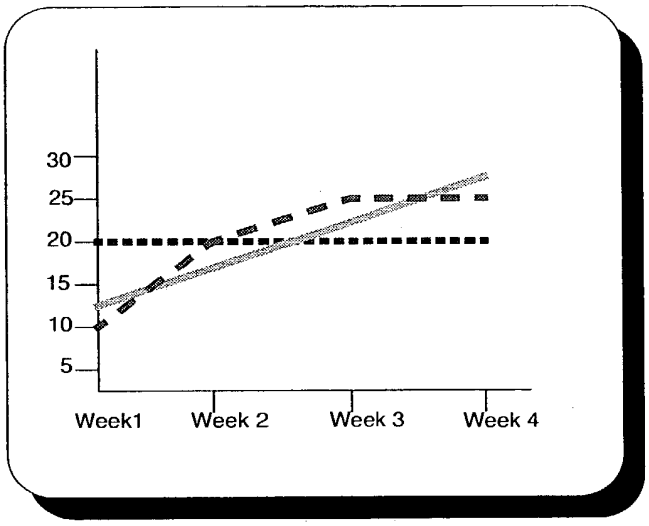


Each product is independently forecasted for order promising purposes. For example, a different forecast exists for computers ordered by Electronix, Inc. and by SRC Computers.

The information (fields) that describe a product include the following:

- Name of the product
- Group(s) to which a product belongs
- Supplier and customer(s)
- Item(s) which fit the specification of the product
- min_quantity, min_delivery_lead_time
- forecast_policy which specifies how to convert a forecast into individual forecast requests. See FIGURE 71 for an example of an input forecast for 80 demand units in the next four weeks. The figure shows three different ways in which the forecast demand can be handled by forecast_policy.

FIGURE 71 Product Forecast Policy



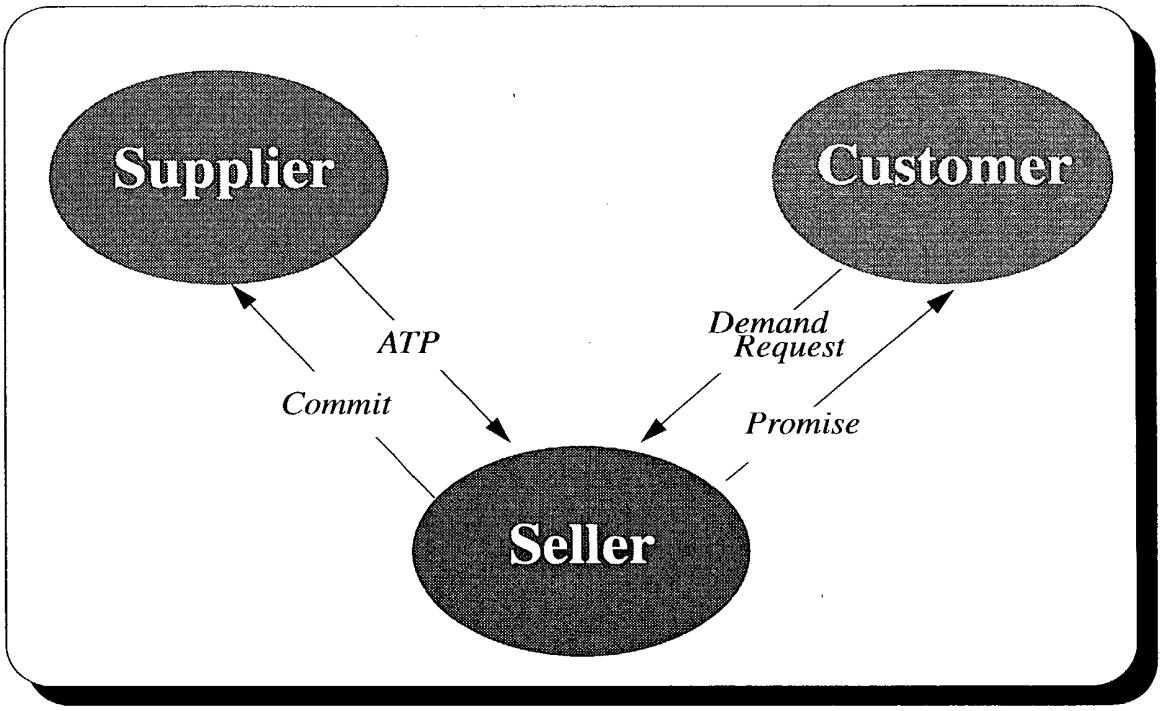
External demand for a product is taken in by:

- Product forecasts
- Product demand

8.8.4 Basic Demand Model

FIGURE 72 shows the basic demand model.

FIGURE 72 Basic Demand Model



The following terms are used:

- Commit - sales commitment, from forecasts, by the seller to a supplier. This is constrained demand. It is the Quantity of this product or product_group that the Seller is willing to commit to selling for specified delivery_dates. This could also be called *requested ATP*. It is the Quantity that will be allocated as available-to-promise for this particular Seller as long as it is feasible to produce. The committed value results in forecast Requests. If those forecast Requests can be met and Promises are made from the supplier Sites, then those Promises make up the ATP. The ATP has been allocated to the Seller, and the Seller can use that ATP to immediately make Promises to actual Requests from customer Sites.
- ATP - supply allocation, or the items the supplier has designated as available to satisfy some or all of the sales commitment, but that have not yet been sent from the supplier to the seller. This is constrained supply. It is the Quantity of this product or product_group for which Promises have been allocated to this Seller for the specified delivery_dates. This could also be called gross ATP, which is the ATP prior to being consumed by actual Promises. The available to promise is this allocated minus the actual Promises that have been made by consuming this allocation. See FIGURE 73.

- Demand Request - the demand, or need for a particular product, made as a request to the seller by a customer.
- Promise - a promise made by the seller to a customer in response to a demand request.

FIGURE 73 Allocation

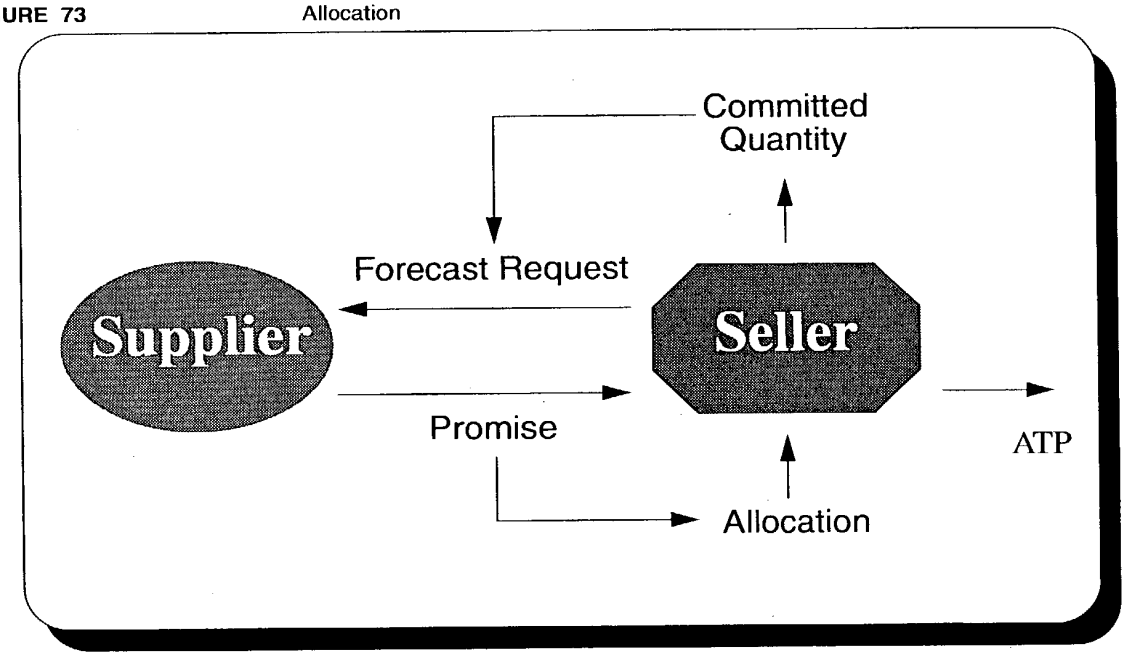
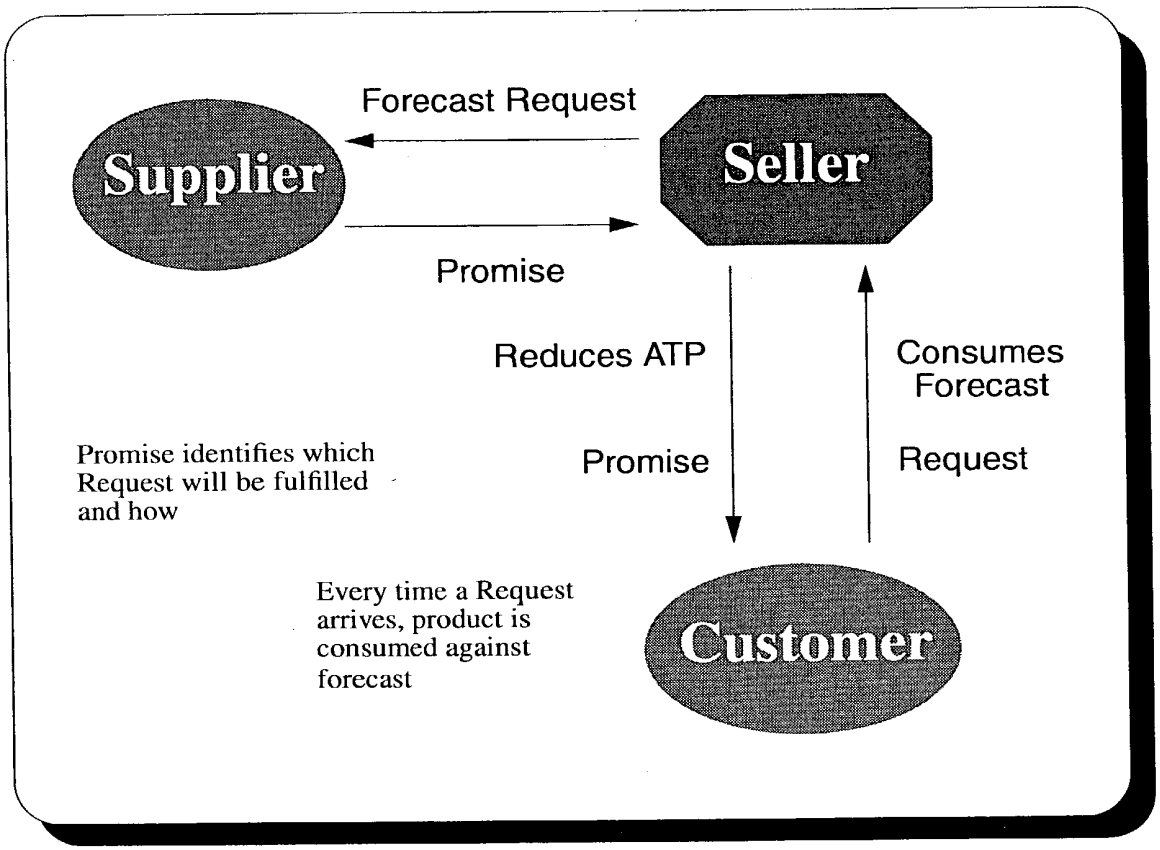


FIGURE 74 shows the forecast consumption process.

FIGURE 74 Forecast Consumption

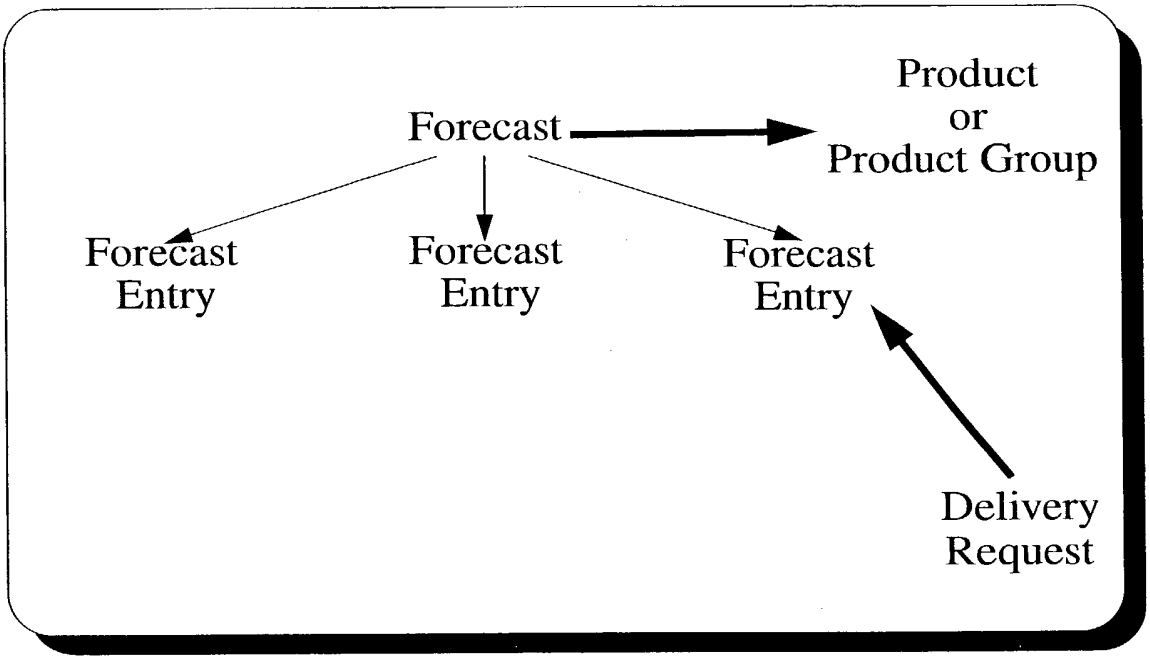


8.8.5 Product Forecasts

In each seller plan there is one forecast for each product or a product group. There is a forecast entry for each product for each time period. A forecast entry may generate multiple forecast requests based on forecast policy.

FIGURE 75 shows the relationships between a forecast, forecast entries, products and product groups, and delivery requests.

FIGURE 75 Product Forecasts



8.8.6 Request and Promise

See the Request and Promise section in this manual for more information about request and promise.

8.9 Display a Table from a Database

A table from a database may be displayed using an axis-cross layout. The worksheet might be:

```
replicating record_list (Worksheet df_wrk)
{
  [ A1 stream = recalc_stream(df_wrk); ]
  [ A2 rec = stream.records; ]

  [ B1 field = stream.fields; ]
  [ B1.title = "Column Names"; ]

  [ C1 = stream.field_value(rec, field); ]
  [ C1.title = "Field Values"; ]
}
```

rec is a list of records, and **field** is a list of column names. C1 is a dependent cell that contains the value of a field of a given record and column. B1 is to be the X-axis and C1 is to be the CROSS. The Y-axis is not important.

To get the following output in the layout:

B1(1)B1(2)....B1(n)
Field Values C1(A2(1),B1(1))C1(A2(1),B1(2))C1(A2(1),B1(n))
. .
C1(A2(m),B1(1))C1(A2(m),B1(2))C1(A2(m),B1(n))

(The above output depicts m records and n columns.)

provide the following layout:

```
axis_cross record_list
{
  worksheet: record_list;

  X: B1;
  Y: A3 CROSS;
  CROSS: C1;
}
```

A2 is needed on the Y axis to get the second dimension. The key field for the record will be displayed (line number). Since the Y axis titles need to be hidden, in the report create another cell dependent on A2 that is always "":

```
[A3 = do(A2, ""); ]
[A3.title = ""; ]
```


then put A3 on the Y axis instead of A2. Then set the width of both A3 and A3.title to 0 so it will not display anything.

8.10 Items in Multiple Products

8.10.1 Description

The advantage of specifying the same item in multiple products is that when a customer asks for that item, the customer gets several quotes that, for example, could vary in delivery lead time and, therefore, in price (soon = expensive, late = cheap).

8.10.2 Quoting Against Multiple Products' Forecasts

By doing the following, you commit to consuming from <product name>:

item_requests.item_promise.consumed_forecast = ...forecasts.find(<product name>)

To see all the options available, you should not do the above. Leave consumed_forecast unspecified and do the following to see a list of all the options:

item_requests.item_promise.item_atp

If you want a specific product, do the following:

item_request.item_promise.item_atp.find(#.product == <product name>);

8.11 Items Sold from Multiple Sites

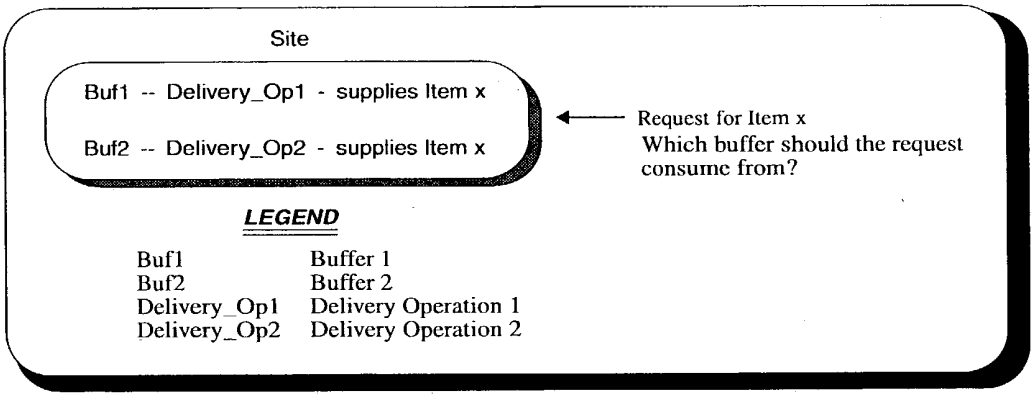
8.11.1 Items

The site specifies the items that other sites can request. Consider an item x which is made at sites 1 and 2. This item is forecasted and sold as products X1 and X2 from sites 1 and 2, respectively. Since both products call for the delivery of the same item, a concern arises when defining the delivery operation associated with the item.

8.11.2 Buffers

If there are two buffers that supply Item x, the question arises as to which one should the external request consume from. See FIGURE 76:

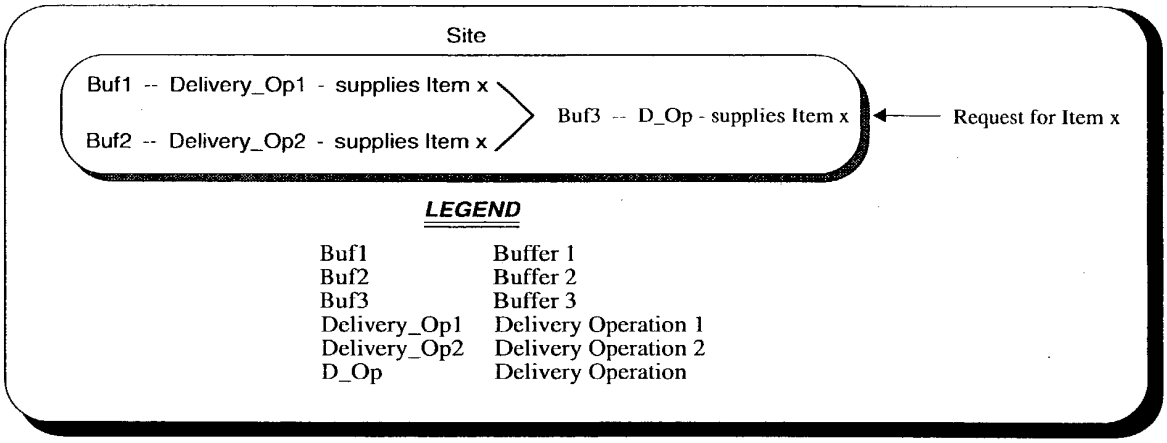
FIGURE 76 Multiple Buffers Supply an Item for a Request



The whole idea of *item.delivery* is to select one buffer from an outside point of view which supplies this item. A request should never have to know the inside details of the site e.g. how may buffers are there. If a request asks for something, it knows one place from which to get it.

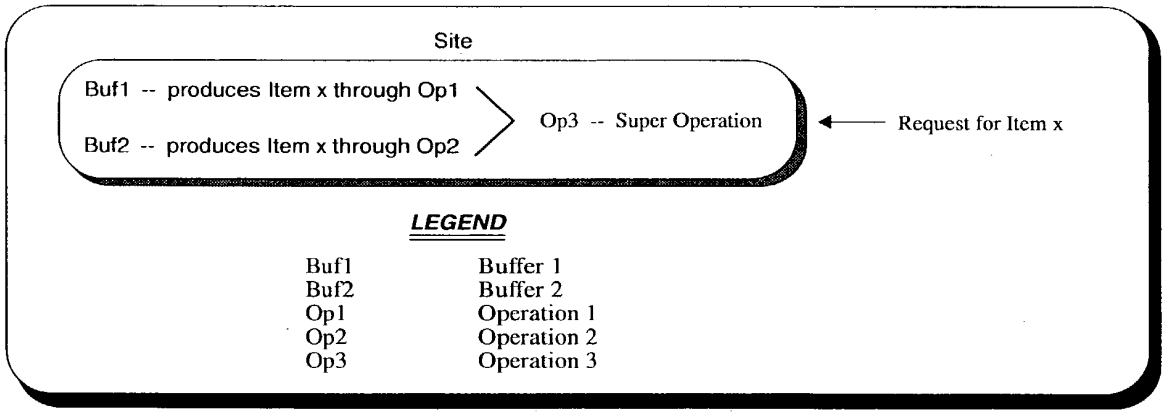
This could be modeled as in FIGURE 77, where two buffers supply the same item (Item x):

FIGURE 77 One Buffer Supplies an Item for a Request



A better way is to not have Buf3 at all. In that case, the model looks as in FIGURE 78:

FIGURE 78 Operation Supplies a Request



8.11.3 Process

The process is as follows:

- Define OP3 as a super operation with OP1 and OP2 as its sub-operations, using the normal super_operation / sub_operation relationship.
- Then use the process extension ALTERNATES_PRIMARY.

- Set *item.delivery* = OP3
- Let OP3 make the decision as to which one to pick OP1 or OP2

A request should not be used to specify things like OP1 or OP2. The whole idea of delivery operation is that from an outside point of view, there is one place a request can go to satisfy itself.

8.11.4 Customer Site

A customer site can choose which supplier site to place a request on, and can choose what item of that site to request. But that customer site cannot dictate decisions within that supplier site. The only thing that the customer site can specify to the supplier site is what the supplier site lets it specify via the item definition. The supplier site could offer item *X-from-Location-1* and item *X-from-Location-2*, such that the customer site gets to specify which he wants. Alternatively, the supplier site could offer Item X with a feature *Location* with options 1 and 2. Then the customer site could specify a configuration that specifies item and location.

If the supplier site only provides a standard Item x, then the supplier site is not giving the customer site the option of specifying location. The customer site has no opportunity to specify where it is delivered from, and the delivery operation will likely be an ALTERNATES_* Operation with sub_operations that deliver from one of the two. The delivery operation will be making the choice from where to deliver (hopefully, intelligently based upon the distances/costs involved).

8.11.5 Modeling One Supplier

To model one supplier site that offers to its customers the choice of *Item X from Location 1* or *Item X from Location 2*, the Location needs to be part of the Item *spec*, either a feature and option of the one item, an attribute of the one item, or have two separate items *X from Location 1* and *X from Location 2*. If it is not an item, then it cannot be a request from a site. Each site must have the ability to specify what it will allow its customers to request.

8.11.6 Modeling Two Suppliers

To model two supplier sites that offer the same item, the customer request can specify the site in the request.

8.11.7 Item Request

The Item_Request model would specify the location from which the requested item should be delivered, resources used, operations chosen, and raw materials used. Item_Request can only specify what the item definition allows it to specify. It could be none, any, or all of the above. The Item *spec* dictates. The attributes must be incorporated into the item definition.

8.12 Nonexistent Parameter

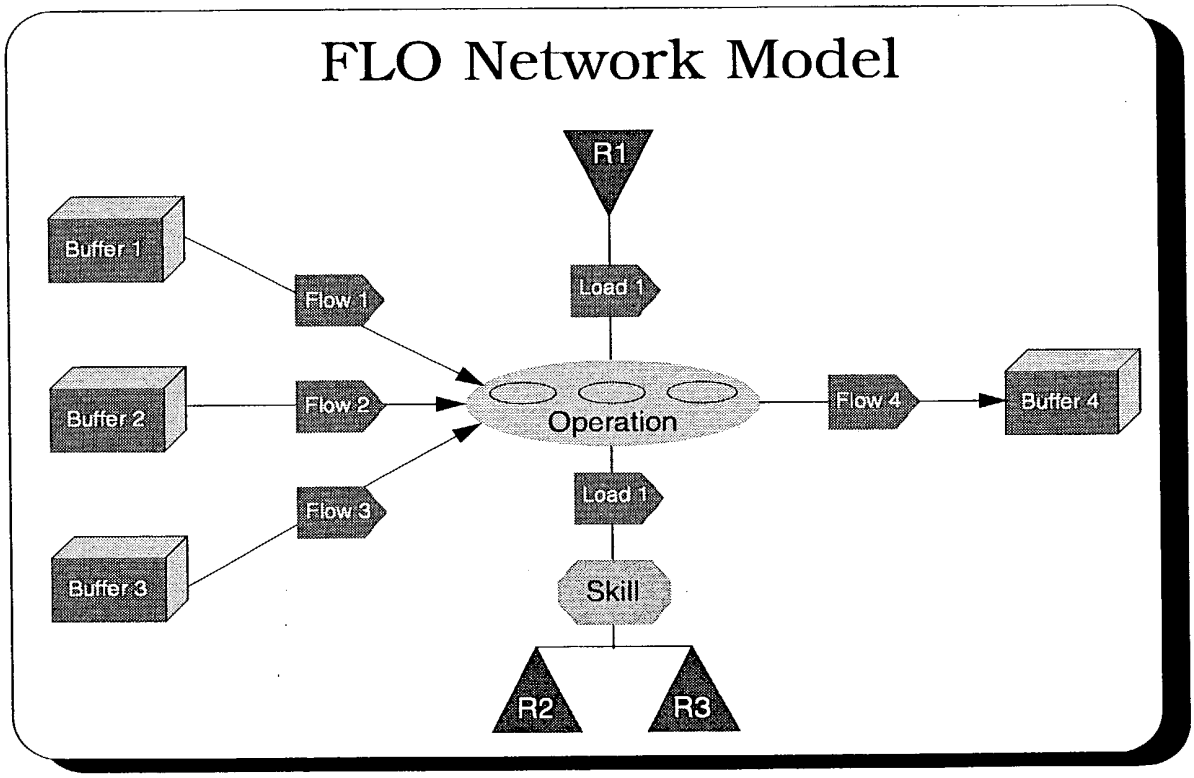
Create a worksheet with a nonexistent parameter and then update it later without the layout getting messed up. Without this, reports take too long to come up since they must fill all the layouts, even ones that may not be selected by the user.

8.13 FLO Network

8.13.1 Description

The FLO forms the network of Flows, Loads, and Operations within a given site. It defines the flow of material through the factory. See FIGURE 79. Each element in this figure represents a model in the *Rhythm*® Model Reference Manual. See this manual for additional details.

FIGURE 79 FLO Network Model

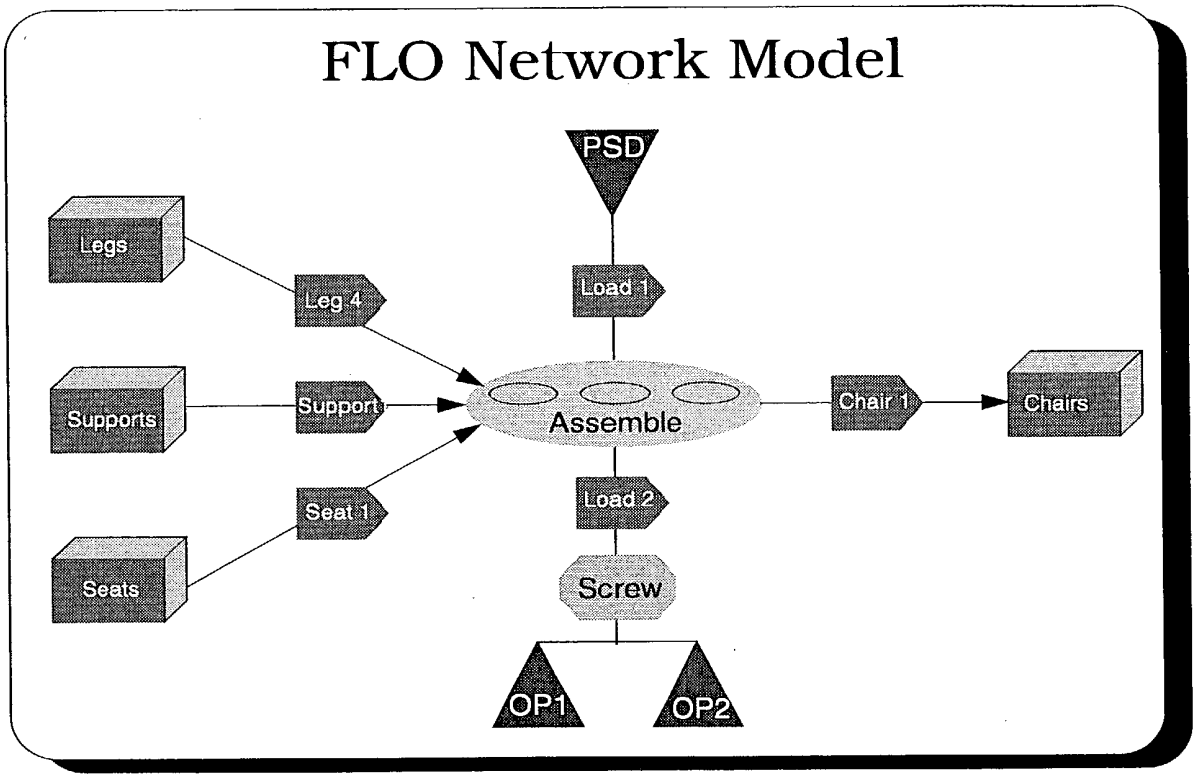


Factories could have different types of flows such as:

- straight line
- assembly
- disassembly
- combinations of each of these flows

Planning becomes more difficult as the complexity of the FLO network increases. See FIGURE 80 for a relatively simple planning scenario. Note the elements of FIGURE 79 to which the elements of FIGURE 80 apply.

FIGURE 80 FLO Network Model - Chair



8.13.2 Operation Model

The operation models the value adding activity. It consumes one or more input items and produces one or more output items. The connecting arc between the buffer and operation is flow. The connecting arc between the skill or resource and the operation is load.

8.13.3 Buffer Model

The buffer models management of material. Each buffer manages the flow of one item. Buffer uses a Flow_Policy extension to implement material planning rules. Buffer has supplying, storage, receiving, and picking, operations.

8.13.4 Flow Model

The flow models how material is used by the operation. It connects buffer to operation, whether flowing from the buffer into the operation or flowing out of the operation into the buffer. The Flow_Policy that is defined is a buffer extension.

Flow has an extension named Usage_Policy. The flow defines how an operation consumes or produces an item through this extension. Example Usage_Policy extensions include:

- Consume_per
- Produce_per
- Consumed_fixed
- Produced_fixed
- Produced_Yield

8.13.5 Resource Model

Resource models the capacity to perform operations. Each resource has a skill group. Examples of resources include:

- machine
- tool
- fixture
- trucks
- operators

A resource has extensions such as:

- Load_Policy
- Efficiency
- Maintenance - defines how maintenance is specified for a given resource.
- Size - defines the size limits on the loads that can be placed on a resource.
- Variability - models the uncertainties and creates pads before and after the operation performed at this resource.

Resource also has operations such as:

- transit_operation
- skill_operation
- setup_operation

8.13.6 Load Model

Load model connects skills or resource to an operation. Usage_Policy is an extension of load which defines how a given operation uses the skilled resource specified by the load. Operations can have multiple loads. They model simultaneous skilled resources.

The Load_Policy that is defined is a resource extension.

8.13.7 Skill Model

Skill models a capability needed to perform an operation. Each skill has a list of resources which have that skill. Each resource has a different efficiency in performing a certain skill. Skill allows the modeling of alternate resources. Skill has an extension called *selection*. It implements the rules for an alternate resource selection. If a skill is specified, the Usage_Policy determines how available resources capable of that skill are to be loaded.

8.14 Releasing Operation Plans

8.14.1 Operation and Operation_Plan Fields

There are several fields in the Operation and Operation_Plan models which apply to releasing operation plans. The following excepts are from the Model Reference Manual.

8.14.2 Operation Model

The following excerpts from the Model Reference Manual describe fields in the Operation model which apply to releasing operation plans.

release_fence - - *a* **Horizon_Date** *field of model* **Operation**
An offset from the "current" date of the plan, by which operation plans should have already been released. Operation plans with a scheduled start which are not released by the release_fence will cause a UNRELEASED problem to be raised.
Default: 0

release_soon_fence - - *a* **Horizon_Date** *field of model* **Operation**
An offset from the "release_fence" of the operation plan, which serves serves as warning that the operation plan should be released. Operation plans not released by the release_soon_fence will cause a NEEDS_RELEASE problem to be raised.
Default: 0

release_name_expr - - *a* **Expression** *field of model* **Operation**
An expression used to calculate the release name to be used for operation plans based on this operation when they are released without specifying a release name. Note that "#" in the expression refers to the Operation_Plan, not the Operation. The default expression is: "if (and(#.top_operation_plan.exists, #.top_operation_plan.released), top_operation_plan.release_name, #.operation.name & \"--\" & #.operation.next_release_number)"
Default: Release name or operation_name & next_release_number -- see details above
Properties:command=True

next_release_number - - *a* **Integer** *field of model* **Operation**
When operation plans are released without specifying a release name, a release name is generated for them, based on an expression (see the release_name_expr field). This expression can contain an operation-specific, unique integer so that the release names for all operation plans corresponding to a particular operation are different. next_release_number provides the next number in the sequence and is intended for use in release_name_expr. Using next_release_number in an expression will cause the internal counter to increment each time it is evaluated.
Default: 1
Properties:Export-Only Field

release_number - - *a* **Integer** *field of model* **Operation**
The last release number generated by next_release_number. When the value is zero, no release numbers have yet been generated. The value may be set, in which case the next release number generated with next_release_number will be 1 greater than the value set. Note that setting this value smaller than the largest value already used may cause generated names to be duplicated.
Default: 0

8.14.3 Operation Model

The following excerpts from the Model Reference Manual describe fields in the Operation model which apply to releasing operation plans.

released - - *a* **Logical** *field of model* **Operation_Plan**
If "true", then this Operation_Plan has been released to be performed. Once released, it is subject to greater constraints. Its motive and quantity will not be changed, and it will be executed as soon as feasible. The material assigned to it is considered "allocated" and will not be "re-allocated". Of course, users may change anything about a released Operation_Plan they desire. When the Operation_Plan is for an operation which is part of an operation hierarchy, (it has sub-operations, or it is itself a sub-operation, or both), the entire hierarchy is released together.
Default: false

release_name - - *a* **Symbol** *field of model* **Operation_Plan**
The 'release_name' used to identify an Operation_Plan when it is being processed. This is often called an "order id". This need not be used -- for example, repetitive manufacturers often do not need to bother with a 'release_name'. But even in that case, a 'release_name' will be generated when 'released' is set "true" (see the description of the release_name_expr field). If the Operation_Plan is was not released when the release_name is set, the Operation_Plan will be released. As with released Operation_Plan's, if the Operation_Plan is part of a hierarchy, the release_name refers to the entire hierarchy.
Default: none
Properties:command=True

8.14.4 Main Points

The operation plan has three points which are important for releasing operation plans:

- Current, which is the current date for planning purposes in the horizon. If you get operation plans before this date which have not been released, you see PLANNED_BEFORE_CURRENT problems. This is an INFEASIBLE problem.
- Release Fence, which is offset from Current, and specifies a date by which operation plans should be released. If you get operation plans before this date which have not been released, you see UNRELEASED problems. This is an INFEASIBLE problem.
- Release Soon Fence, which is offset from Release Fence, and specifies a date to warn the user that operation plans need to be released. If you get operation plans before this date which have not been released, you see NEEDS_RELEASE problems. This is a FEASIBLE problem.

As you can see, UNRELEASED, and NEEDS_RELEASE problems are nested. If an operation plan has an UNRELEASED problem, it necessarily has a NEEDS_RELEASE problem as well. Likewise, PLANNED_BEFORE_CURRENT problems imply UNRELEASED problems. To avoid over-information, only one of these problems will be associated with a particular operation plan at any given time. The problem retained will be the most severe of the three. The others, if any, are automatically removed. Note that they are removed. Their resolvers are not applied to the problem. So, if you leave the *release_fence* and *release_soon_fence* at their defaults, you will only see PLANNED_BEFORE_CURRENT problems.

8.14.5 Notes About Released Operation Plans

The following notes apply to released operation plans:

- An operation plan is defined as released if-and-only-if it has a release name.
- When an operation plan supplies an LFL buffer and that operation is released, then the corresponding consuming operation is automatically released.
- PLANNED_BEFORE_CURRENT is a feasible problem.
- Releasing the consuming operation causes the upstream supplying operations to be released.

8.14.6 Releasing Operation Plans

An operation plan can become released in a variety of ways:

- The user assigns it a release name. In OIL, this would be:
  opplan.set_release_name("SOME RELEASE NAME")
- The user explicitly releases it. In OIL, this would be:
  opplan.set_released(true)
  Doing it this way causes a release name to be automatically generated for the operation plan. See Generating Release Names.
- The resolver for the NEEDS_RELEASE problem releases the operation plan.

If a released operation plan is subsequently unreleased by a user action (*opplan.set_released(false)*), its release name is removed. You cannot set a *release_name* to the empty string to un-release an operation plan. You must use *set_released(false)* to un-release a released operation plan.

For operation plan hierarchies (routings, alternates) the entire hierarchy is released together. In this scheme, the entire hierarchy will have the same release name. The *release_name* is stored in the top operation of the hierarchy. That is why you see references to *top_operation_plan* in the *release_name_expr*.

8.14.7 Generating Release Names

8.14.7.1 Release Name Expression

When an operation plan is released by means other than setting its release name, Rhythm will generate a release name for it. This name is computed using the *release_name_expr*, which is a field of the Operation model.

The default *release_name_expr* will use the release name calculated for the topmost operation plan in the hierarchy. If there is no release name yet (which is the case when we set the release name of the topmost operation plan in an unreleased hierarchy), the name is generated using the operation's name, suffixed with an auto-incrementing, positive integer (the *operation.next_release_number* field function).

Note: The *release_name_expr* is a property of the operation plan (but a field of the Operation Model). The operation plan must exist before you can set the *release_name_expr*. This means you must satisfy all requests and attach the expression to the appropriate plans. As with the other released attributes, when an operation plan is part of a hierarchy, the attribute applies to the entire hierarchy. So, setting the *release_name_expr* for an operation plan applies to all operation plans in that hierarchy.

8.14.7.2 Release Number

To facilitate unique release names, the operation provides a release number. This allows all plans generated from that operation to be associated with a unique positive integer. The numbering is local to plans derived from a particular operation. To get the next number in the sequence for an operation, you use the *next_release_number* field function of the Operation model. Each time this function is called, the counter increments. If you want to know the number last generated, use *release_number*. The *release_number* field can also be set, so you can create holes in the auto-numbering.

WARNING: DO NOT SET THE NUMBER TO LOWER VALUES. If you do, the whole purpose of generating unique names will be circumvented.

8.14.7.3 Example

If you have a plan generated based on operation *My-OP* and its name is generated (with the default expression in place for those operation plans), the release name will be *My-OP--1*. The next time an operation plan based on *My-OP* is released, it will be named *My-OP--2* and so on. Operation plans based on *Your-OP* which have their release names generated will start out with *Your-OP--1* and go from there. Note that the numbering is associated with the releasing of the operation plans, not the creation of the operation plans themselves.

Note that the release numbers and release expression are part of the Operation model. So, in the release name expression, you need to do something like the following to get the next release number for plans generated from that operation:

#.operation.next_release_number

8.14.8 Fences, Problems, Resolvers, Strategies

The *release_fence* marks a time, offset from the current date of the plan by which operation plans should be released. When you specify the fence, it is considered an offset from the current date of the plan.

The *release_soon_fence* marks a time, offset from the *release_fence* of the plan and serves as a warning to the user that certain operation plans are in need of being released.

When an unreleased operation plan begins prior to one of these fences, the corresponding problem is created, as described above. The resolver for UNRELEASED will perform a move-out, in the same way that happens for PLANNED_BEFORE_CURRENT. The move-out will be to the current release_fence. The resolver for NEEDS_RELEASE will release the operation plan, generating the name as specified by the *release_name_expr*.

When current moves, so do the release and *release_soon* fences. Moving of current will cause detection of PLANNED_BEFORE_CURRENT, UNRELEASED, and NEEDS_RELEASE problems, and the corresponding removal of the nested problems as described earlier.

Note that strategies focused on INFEASIBLE problems will not catch NEEDS_RELEASE problems (because it is a FEASIBLE problem).

8.14.9 Buffers and the Releasing of Operations

There is no relationship between buffers and the releasing of operation plans. Releasing of an operation plan affects the operation plan (and its hierarchy if there is one), without replanning and propagation. The only thing that occurs besides changing the state of the operation plan when it is released is the status of three problems (PLANNED_BEFORE_CURRENT, UNRELEASED, NEEDS_RELEASE) are updated.

Note: The problem status is updated. The problems are not automatically resolved.

8.15 Replanning

Changes that require replanning include:

- Importing new yield numbers - plan demand, satisfy all requests, show loads. To examine a yield crash in a certain operation, import the new yield numbers for the relevant operations and then replan all operations to reflect the changes.
- Change in request quantity
- Change in date
- Change in on-hand inventory / WIP in a buffer
- Change in available capacity

8.16 Request and Promise

8.16.1 Description

Request and promise is the mechanism by which different authority domains talk to each other. The request and promise mechanism performs the following functions:

- Turns forecasts into available-to-promise
- Implements actual demand and its fulfillment
- Handles communication between sites in separate authority domains

The same request mechanism is used by forecast and actual demand (orders).

FIGURE 81 shows the relationships between the parts of the request model. The request model has the following structure:

- Request - has a list of Delivery_Requests (one or more items to be delivered on a given date)
- Each Delivery_Request has a list of one or more Item_Requests
- Each Item_Request contains the quantity of the item and the date(s) requested.

FIGURE 82 shows the relationships between the parts of the promise model, and shows how a promise model is similar to a request model.

FIGURE 81 Request

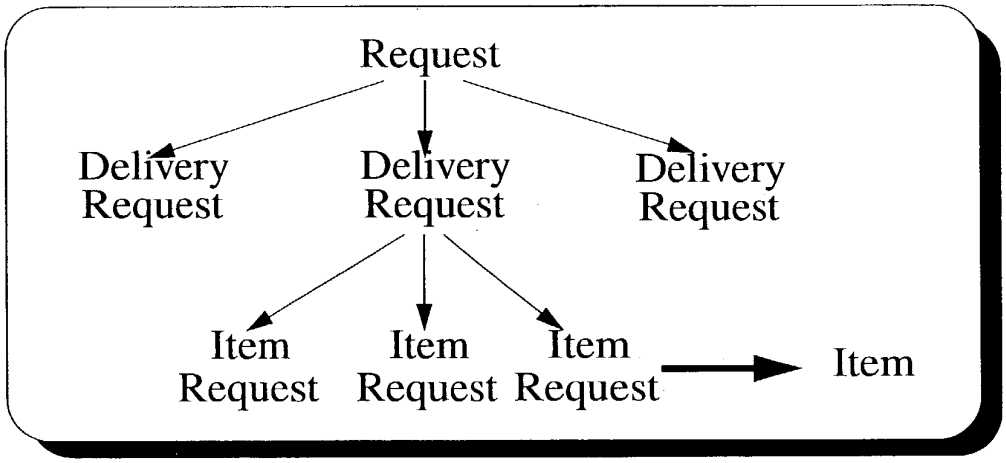
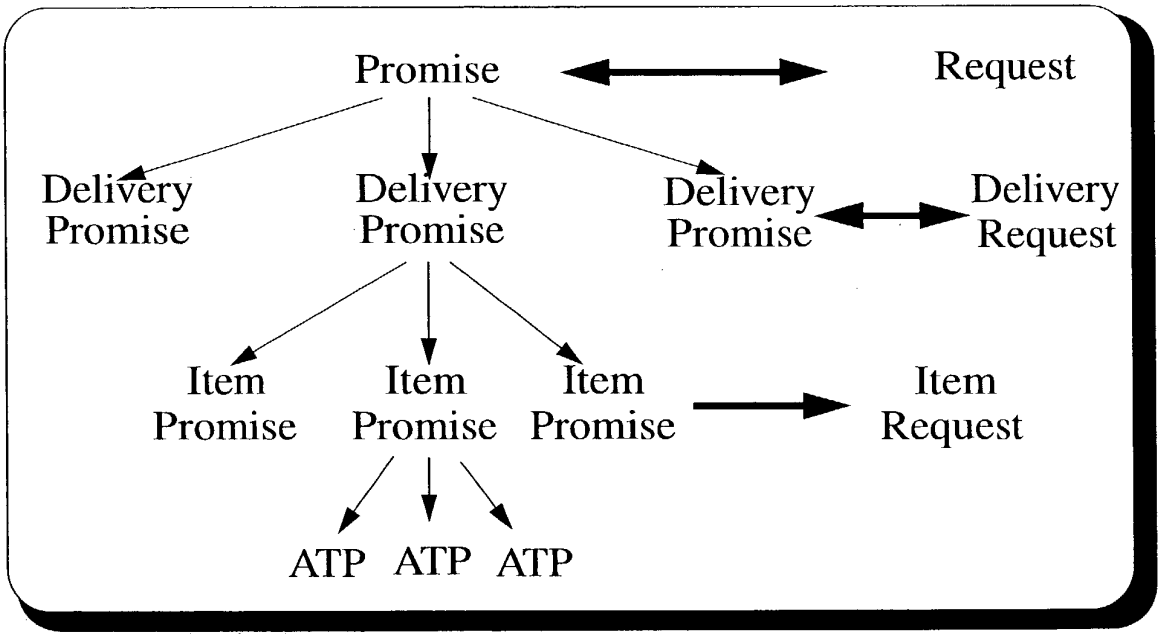


FIGURE 82 Promise



8.16.2 Request and Promise Procedures

The Request, Delivery_Request, and Item_Request models together model requests from one site to another. The Promise, Delivery_Promise, and Item_Promise models together model the supplying (promising) site's commitment back to the requesting site. The request and promise procedures consists of the rules as described next.

When a request is issued by Site A (See FIGURE 83), then Site B:

- Offers a promise (which may differ from the request)
- Rejects the request

If Site B offers a promise, then Site A:

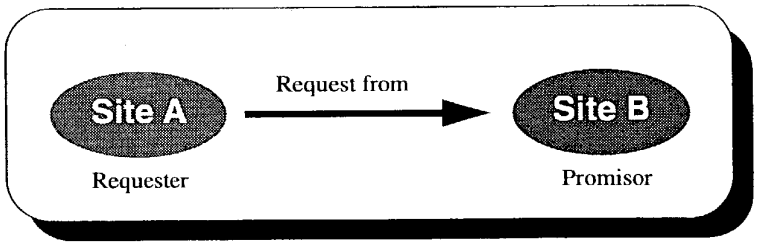
- Accepts the promise
- Deletes the request
- Reissues a modified request

If Site B rejects the request, then Site A:

- Deletes the request
- Queues the request for future consideration by Site B
- Reissues the request, possibly modified

When a request is an actual order, only the customer should be modifying the request and only the suppliers should be modifying the promise.

FIGURE 83 Request and Promise Procedure



8.16.3 Policies

Policies for handling requests and promises include the following:

- The promise is good until a specified date
- The request is good until a specified date

8.16.4 Registering a Request with the Active Plan

Just because a person requests a refrigerator to be built does not mean someone will actually plan to build a refrigerator. That is a decision to be made. Thus, the request is registered upon creation, without a plan to satisfy the request being created.

The *plan_to_satisfy* field indicates the decision to create a plan for the request that has been made. A plan is created to satisfy the request. Until then, the request is unplanned.

When a request is created, its associated promise is also created.

8.16.5 Promising a Due Date

If the plan satisfies the request, then another decision must be made. Just because a refrigerator can be built does not mean someone promises to build it.

Thus, the *promise_as_planned* field indicates the decision to promise to perform the plan that has been made. At that time, the state of the promise is changed to indicate a promise of the due date that is planned.

8.16.6 Planning a Request in Due Date Order

There is no need to plan requests in due date order, though it is not unreasonable to do so. To plan requests in due date order, get the list, sort it, and then plan the requests.

8.16.7 Accepting a Promise

When a due date (plan) has been promised, the promise has been offered. The handshake is not complete until the requestor accepts the offered promise. At that point, the handshake is complete (i.e. a request has been made and a promise given), and both sides are committed.

The promise is accepted by setting the *accepted* (date) field on the Request Editor.

8.16.8 Planning to Satisfy

Consider that 50 items have been promised for May 1. The request is then changed to 60 items needed for May 2.

In that case, 50 items are committed, but 60 items should be considered. Planning should be done for one or the other. Planning to satisfy the commitment of 50 items should be done, or planning to satisfy the new request should be done (to determine if the quantity needed can be promised).

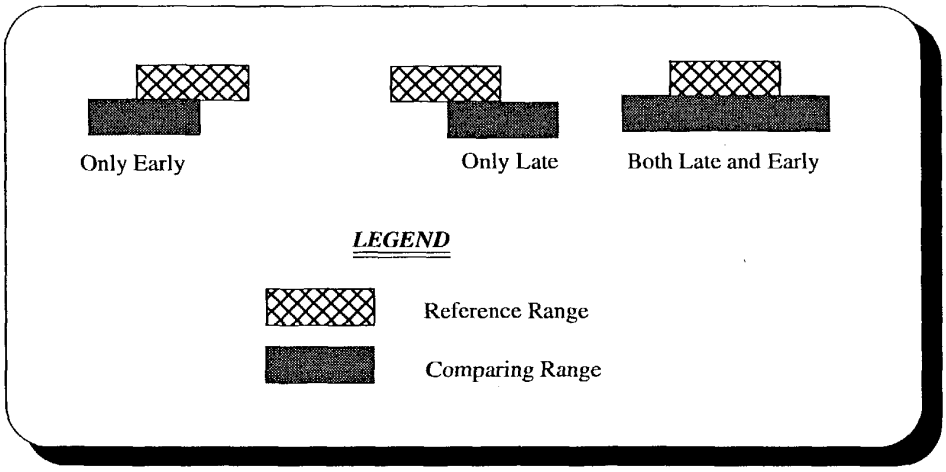
So, *plan_to_satisfy* should be used for the request for 60 items on May 2 to try to plan to meet the request. If successful, then *promise_as_planned* should be done. If not successful, *plan_to_satisfy* should be used for the promise for 50 itmes for May 1 to go back to planning to satisfy the promise, which is still binding.

8.16.9 Range Overlap

In all request and promise problems, one date range is compared with another for lateness, earliness, shortness, or excess. FIGURE 84 shows range overlap. In the figure, reference range is the requested date range, and comparing range is the promised date range.

Late or excess problems exist when the maximum of the comparing range is greater than the maximum of the reference range. Early or short problems exist when the minimum of the comparing range is less than the minimum of the reference range. For example, if the requested date for a request is May 1 - 3, and the promised date is April 30 - May 4, then the promised date range is both greater than and less than the requested date range. Hence, both excess-short or late-early can co-exist. In request and promise, only short or late are shown when both excess-short or late-early exist. See FIGURE 84. The resolver for one eliminates both if the other one existed or was created while resolving.

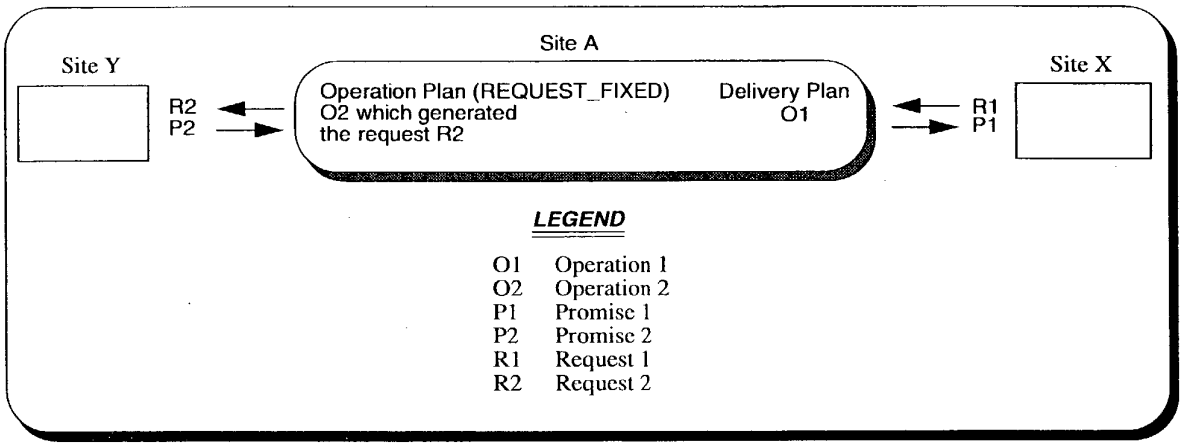
FIGURE 84 Range Overlap



8.16.10 Request and Promise at a Site

FIGURE 85 shows how a request and promise looks on a site.

FIGURE 85 Request and Promise on a Site



Request R1 is placed on Site A by one of the following:

- Actual customer
- Seller (through forecast policy)
- Another site (through REQUEST_FIXED operation)

A request keeps the information such as when it is was issued and whether or not it has been promised.

Each Item_Request creates a plan for the quantity and dates as specified in the Delivery_Request.

Site A plans for the quantity and/or offers a promise P1 against the request R1. Site A is modeled to procure material from other sites, so it can place request R2 on another site.

8.16.11 Problem Types

8.16.11.1 Description

This section discusses the types of problems that can occur with requests and promises. It uses the example shown in FIGURE 85 as a basis for the discussion.

There are two types of problems for Site A as listed in Table 16. The two types of problems are about the relationship between request and promise pairs R1 and P1, and R2 and P2. Within the two types the problems are further divided into five blocks, which are described later. Note that each problem is a *category* as defined in the Problem_Set model. Refer to the *Rhythm* SCP Model Reference manual for more information about the Problem_Set model.

Table 16: Problem Types for Site A

	Type 1: Problems between, or due to, R1 and P1	Type 2: Problems between, or due to, R2 and P2
B1	REQUEST_NOT_PLANNED	
	PROMISE_NOT_PLANNED	
B2	REQUEST_PLANNED_LATE	SUPPLY_PLANNED_LATE
	REQUEST_PLANNED_EARLY	SUPPLY_PLANNED_EARLY
	REQUEST_PLANNED_SHORT	SUPPLY_PLANNED_SHORT
	REQUEST_PLANNED_EXCESS	SUPPLY_PLANNED_EXCESS
B3	PROMISE_PLANNED_LATE	
	PROMISE_PLANNED_EARLY	
	PROMISE_PLANNED_SHORT	
	PROMISE_PLANNED_EXCESS	
B4	REQUEST_PROMISED_LATE	SUPPLY_PROMISED_LATE
	REQUEST_PROMISED_EARLY	SUPPLY_PROMISED_EARLY
	REQUEST_PROMISED_SHORT	SUPPLY_PROMISED_SHORT
	REQUEST_PROMISED_EXCESS	SUPPLY_PROMISED_EXCESS
	ITEM_PROMISE_OVERPRICED	
	DELIVERY_PROMISE_OVERPRICED	

Table 16: Problem Types for Site A

	Type 1: Problems between, or due to, R1 and P1	Type 2: Problems between, or due to, R2 and P2
B5	PROMISE_NOT_OFFERED	
	PROMISE_NOT_ACCEPTED	
	REQUEST_QUEUED	

8.16.11.2 Supply Problems

Supply problems can occur for requests that are generated through the REQUEST_FIXED process. (These are requests from one site to another.) These problems exist on the site that generated the requests. For each supply problem, there is an equivalent problem on supplying site. However, the resolvers are different for each site.

8.16.11.3 Problem Levels

Due to the structure of the request, various problems could exist at the different levels of the Request model. See Table 17.

Table 17: Problem Levels

Problem	Model (for which the problem may exist)
Procedure Problems (B5)	Request
Date (late / early)	Delivery_Request
Quantity (short / excess)	Item_Request

Type 1 problems (between, or due to, R1 / P1) can be further sub-divided into blocks B1 to B5. Table 18 shows each block and its associated problem types.

Table 18: Problems Due to R1 P1

Block	Problem
B1	Not planned problems
B2	Discrepancies between request and the plan
B3	Discrepancies between promise and the plan
B4	Discrepancies between request and promise
B5	Procedure problems which inform the user of the state of the request

At any point time, there is only one plan. It can be either for the request or for the promise, never for both. Hence, problems from B2 and B3 can not co-exist. Both a request as well as a promise can exist in the system, but the plan is only for one of them (or none).

8.16.11.4 Not Planned Problems

B1 (not_planned) problems are more involved. What it means for a request or promise to *exist* must first be defined. As previously described, a request for some quantity and date is created by an actual customer order, a seller forecast, or another site. At the same time the request is created, a corresponding promise is generated. When the promise is initially generated it is for 0 quantity and an infinite future date. This is equivalent to not having a promise; the promise does not *exist* as far as SCP planning is concerned. A PROMISE_NOT_PLANNED problem does not exist for such a promise. After a user fills in quantity and date for this promise, it then becomes plannable. Hence, a PROMISE_NOT_PLANNED problem can then exist, because the promise now exits but it has not yet been planned.

Offering a promise means that the supplying site has committed a specified quantity. It could be 0 quantity and infinite future date. When the quantity offered in a promise is 0, the request is not used for planning, planning_problems, or *not_planned* problems.

A REQUEST_NOT__PLANNED problem exists when no plan exists or the plan is for a promise, and the request is overriding.

A PROMISE_NOT_PLANNED problem exists when no plan exists or the plan is for a request, and the promise has a non-zero quantity and a non-infinite future date.

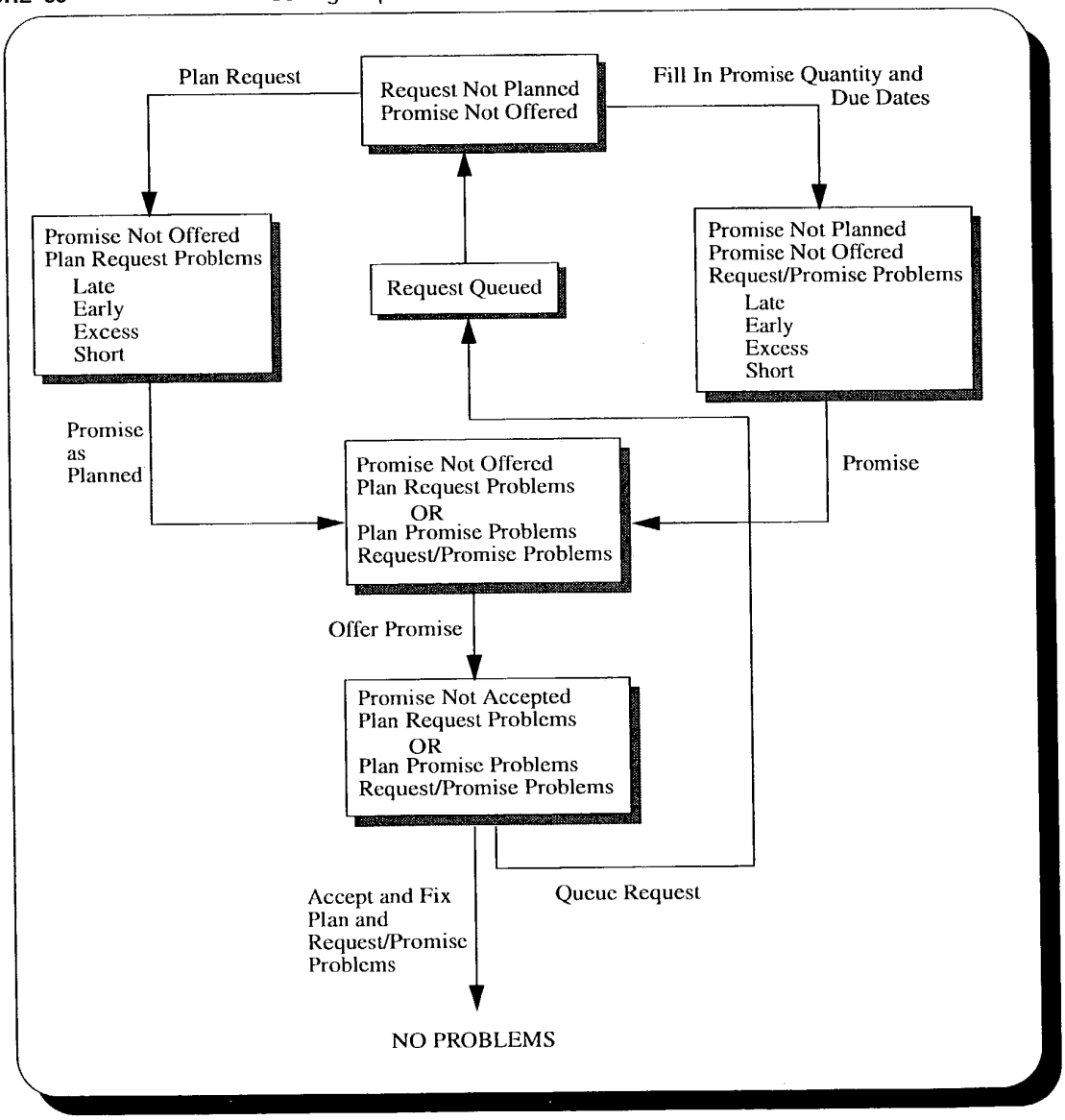
Each unplanned request generates a REQUEST_NOT_PLANNED problem for each item request and a PROMISE_NOT_OFFERED problem for each request.

A request is defined as overriding if the date that the request is made is after the date at which the most recent change was made to this promise, and the promise exists (or was offered). This date is until the promise is not offered (the date that the request is made is after the date that the supplier promised delivery to the customer).

8.16.11.5 Solving Problems

FIGURE 86 shows the various request and promise problems that can occur, and the actions taken by a user or the SCP planner to solve the problems.

FIGURE 86 Solving Request and Promise Problems



8.16.12 Request and Promise Problem Resolvers

There are three basic types of request and promise discrepancy problems:

- Between plan and request or plan and promise, on supplying site
- Between request and promise, on supplying site
- Between promise and the receiving site request or plan and the receiving site request, on receiving site

To find out what exactly these problems are and when they exist, see the sub-section 8.16.10 Request and Promise at a Site.

The resolvers for these classes of problems are as follows:

- There is a discrepancy between what is planned and what the actual request or promise is. In this case, request or promise is considered to be the master, and the delivery plan moves or resizes if the strategy focus allows that.
- The promise offered in response to a request is different in quantity or due date. In this case, the request is considered to be the master, and the promise dates/quantity are set to whatever is requested. NOTE: If there already is a plan for this unsatisfactory (short/excess/late/early) promise, then this resolver generates *promise_planned* problems.
- The supply problems are similar to the ones above. The only difference is that they are on the receiving site and not the supplying site. These resolvers are different from the resolvers of the corresponding supplying site problems. Instead of changing the delivery plan or the promise, these problems would consider these to be the master, and change the receiving operation plan.

These resolvers resize/move the operation plan which supplies them or receives the item. This creates another problem on that buffer. In certain limited capacity situations, these resolvers might be working against each other. In this case, the user should set the strategy such that it does not resolve the request problems.

By resolving supply problems, some late orders may occur at the end and might be the best that can be done with the existing resources. Therefore, it should be accepted. If this situation is acceptable, the strategy should be told to ignore late order problems.

The resolvers for request problems try to shift the plan back which puts more load on the resources in a very limited capacity situation. It might not be possible to balance the load, so the capacity may need to be increased.

8.16.13 Procedure Problems and Resolvers

Some request and promise problems are due to things such as capacity shortage or resource inavailability. There are also problems due to procedures. These problems occur because the procedure has not been completed, e.g. Request_Not_Planned, Promise_Not_Offered, Promise_Not_Accepted. These problems inform the planner of the state of the requests and promises. A planner eliminates these problems in the normal course of planning. If left to strategy, the resolvers for these problems perform the action to resolve this problem and generate the next one in the sequence. For example, note the following resolvers:

- REQUEST_NOT_PLANNED.resolve - this resolver plans the request
- PROMISE_NOT_OFFERED.resolve - this resolver promises according to the plan which leads to a state where promise is not accepted
- PROMISE_NOT_ACCEPTED.resolve - this resolver accepts the promise as it is

8.17 Setting On Hand

8.17.1 Description

Suppose that you need to model a purchasing order (PO). You know that on date MM_DD_YY hh:mm:ss you will receive a shipment of some item. You want to add this PO amount to the on_hand quantity in a specific buffer at this date. Operations that are planned after this shipment can consume from the on_hand quantity.

8.17.2 Assigning Fields

Field assignment can only be done when the field is not read-only by prefixing the field name with *set*. For example:

operation.set_on_hand

8.17.3 set_on_hand

set_on_hand is primarily intended for use when first implementing Rhythm. Rather than import the entire history of everything that has ever flowed in and out of a Buffer, you can say "I went out and looked at the Buffer on Date X and there were N parts in it then." The idea is that, whatever plans are created or moved around after you report that, the Buffer needs to adjust itself so that the on_hand quantity it reports on Date X is exactly N. It is not an inflow of material on that date, it is just a datapoint that tells Rhythm what all the flows are relative to.

Note that this necessarily entails some back-propagation. For example, if you say "On 9/1/96 0:0, there were 100 units there", and later create 30 unit Flow_Plans consuming from the Buffer on 8/15/96, you have effectively said "I consumed 30 units on 8/15, and afterward, there were 100 units." The only possible explanation is that there were 130 units in the Buffer before 8/15.

You can model it by using the Operation_State model to create an Operation_Plan that supplies the Buffer on the date you want.

The on_hand quantity is essentially a function of time. That function has a certain shape, determined by the flows in and out of the Buffer. But it also has a height, which serves as a constant offset from zero. Setting the on_hand quantity never changes the shape of that function, but it does change the height.

8.17.4 Example

Suppose you do not set the on_hand quantity to start with (so it defaults to '0 on_hand quantity at ------'). Then some Flow_Plans are planned through the Buffer. Now you have this:

9/1	+100	100
9/5	-40	60
9/10	+30	90
9/15	-90	0

This Buffer_Plan has the property that the on_hand quantity after the second Flow_Plan is 40 less than the on_hand quantity after the first Flow_Plan. (And that the on_hand quantity after the third Flow_Plan is 30 more than that, and that the on_hand quantity after the last Flow_Plan is 90 less than that.)

Setting the on_hand quantity will never change any of those properties. It will just change the offset. So, for example, suppose you set an on_hand quantity of 80 at 9/7. Well, if there were 80 on 9/7, and a supplier of 30 on 9/10, then the on_hand quantity after 9/10 must be 30 + 80 = 110. Similarly, if there were 80 on 9/7, and the nearest thing before that was a consumer for 40, then the on_hand quantity before that consumer can only be 80 + 40 = 120. Going all the way through, you end up with the following:

9/1	+100	120
9/5	-40	80
9/10	+30	110
9/15	-90	20

It did not change the way consumers and suppliers add and subtract, and it did not create additional Flow_Plans. It just adjusted the offset so that, given the current set of Flow_Plans, the on_hand quantity on the date specified is indeed the value you specified.

8.17.5 Set On Hand Offset

There can only be one set_on_hand offset at any given time. (That is one reason it is not good for modelling POs.) If you set_on_hand for another date and quantity, it has to adjust the offset to match the new setting, which necessarily means abandoning the old one. So, in the example above, if you later do another *set_on_hand*, in this case for 10 units on 9/17, it will re-adjust to the following:

9/1	+100	110
9/5	-40	70
9/10	+30	100
9/15	-90	10

If you import on_hand quantity on 8/15/96 which is your current date to start off with, then change current to 9/1/96 and report the on_hand quantity on that day.

Rhythm abandons the old set_on_hand value and re-plans according to the new one. Just like any other field, when you change the setting, it forgets the old setting and starts using the new one. The only thing that is different is that the field is actually two pieces of data:

- one to specify the date
- one to specify the quantity at that date

If you specify "On 8/15/96 0:0, there were 100 units there" and "On 9/1/96 0:0, there were 130 units there", the system does not create the flows that caused this increase in on_hand quantity.

8.17.6 Flow Plans

Setting on_hand quantity never creates, deletes, or modifies Flow_Plans. You can not say "on 8/15/96 0:0, there were 100 units and on 9/1/96 0:0 there were 130 units" any more than you can say 'the *max_operation_count* of this Resource is 5 and it is 8'. On_hand quantity is not a submodel. It is just a field.

Similarly, if you specify "On 8/15/96 0:0, there were 100 units there" and "On 9/1/96 0:0, there were 50 units there", the system does not create the flows that consumed the difference.

8.18 WIP Assignment

The automated planning function has the capability to accept WIP at alternate parts. The WIP reporting from the external system contains information on operation name, quantity, and time. Rhythm takes in this information and compares it with projected WIP from the plan it generates. The result of the comparison is as follows:

- Reported WIP is less than planned WIP:

  Action options:

  - Check alternate parts
  - Start new wafers
  - Leave it short (planner could choose one of the first two)
- Reported WIP is more than planned demand:

  Action options:

  - Process the WIP and store it in the next stopping point which is either die bank or FGS
  - Other orders should be able to use up the excess

Section 9 Strategy Driven Planning

9.1 Introduction

Strategy Driven Planning is a framework within which a planner can configure the automated planning engine to abide by specific business rules, priorities, and objectives. Through the strategy, the planner specifies which goals to achieve and how to go about achieving those goals. Then the strategies are used in conjunction with the principles of Problem-Oriented Planning (see page 5) to compose an optimal business plan.

9.2 Strategy Driven Planning Versus Just-In-Time (JIT) Planning

9.2.1 Description

Just-In-Time planning is mainly concerned with the following:

- "Getting the pipeline filled at startup".
- Ensuring that supply arrives in time to cover consumption.
- Transferring quantities to avoid creating excess on-hand conditions yet maintaining sufficient quantities to cover consumption.

This presents the problem of needing to manage the amount of product in the pipeline, as well as the timing of moves.

Strategy Driven Planning is mainly concerned with the following:

- Identifying the problems.
- Cataloging the problems.
- Editing the plans using the plan change categories of *move_in* or *move_out*, *resize*, *use_more* or *use_less*, *use_alternate_operation*, *use_alternate_resource*, or *hint*.
- Allowing temporary infeasibility in order to achieve goals.
- Evaluating the plan - making trade-offs and using prudence.
- Moving toward feasibility and desirability.

This presents the situation that enables automated planning to be combined with human planning to attain optimal (the most desirable) plans.

9.2.2 Performing Strategy Driven Planning

Table 19 provides a guide to follow when performing Strategy Driven Planning:

Table 19: Strategy Driven Planning Step-by-Step

Step	Operation	Description and Examples
1	Specify the Problems to Address	The objective is to determine which problems are most important to control. Through a strategy, tolerances can be set for each problem, thus allowing small problems to be ignored. The strategy can also specify the domain to address. For example, which orders or resources, in which problems should be addressed. It can attack one resource, all resources and inventories at a location, etc. Examples of problems are as follows: over-utilized resources, inventory below safety stock, early orders, etc.
2	Specify *Goal*	The objective here is to define the *Goal* for the company. While the strategy is resolving identified problems, it also will try to implement plans to do so which are most desirable. Therefore, the human planner must consider what actions are most desirable. Examples of *Goal* oriented strategies are to either minimize late or early orders, or to minimize inventory or carrying costs.
3	Specify Allowed Adjustments	The objective here is to determine which adjustments are allowed in order to attain a goal. When the automated planner is attempting to eliminate a problem, planning adjustments might be feasible in order to attain a certain goal. Adjustments can be enabled or disabled in the event that a goal cannot be achieved with the adjustments allowed. Examples of allowed adjustments are as follows: an operation being planned earlier or later, quantity being reduced, etc.
4	Specify Time to Quit	The objective here is to define the time to allow for a strategy to complete its function. Termination can be based on several things, such as optimal measure, a problem list, items that have been tried, or time. Examples of termination times are as follows: when a plan close enough to the *Goal* has been found, or if the strategy has gone long enough (time of day).

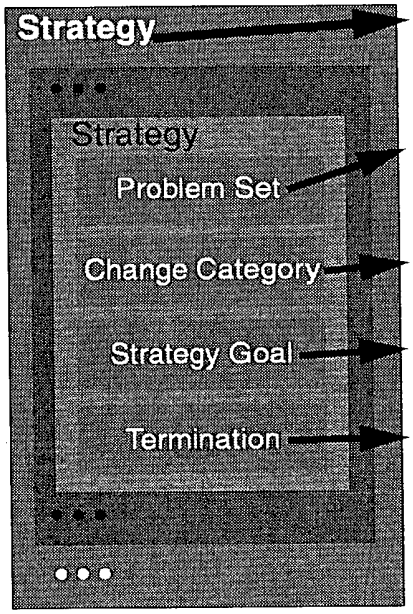
9.3 Overview

This section describes the key topics that are addressed by *Rhythm*® Strategy Driven Planning (SDP). Strategies are models that direct policy-based problem solving algorithms. Strategies specify the following:

- Problems on which to work - categories, allowable tolerances, by location, at all resources
- What is desirable and feasible - operational, financial, other measures, trade-offs between measures, changing scope of allowable plan adjustments
- Times for human intervention - reality checking, business policy enforcement
- Multi-phase strategies - automated through all phases, some phases automated and some phases manual.
- When to stop computing - adequate result, sufficient computing effort.

9.4 Definition

Strategy is defined by the following rules:



Each strategy can have sequenced sub-strategies which can have sequenced sub-strategies.

The problem set defines a set of problems to be addressed by this strategy. Problems are specified in category and tolerances in a horizon. For example, the problem set can be a resource problem, negative on-hand problem, etc.

The change category defines how the problem can be resolved through different ways chosen based on the weight. For example, to avoid lateness, move-in can be used, but move-out should be disallowed.

The strategy goal defines a goal and a target to be achieved by this strategy. For example, the goal is to reach feasibility.

The termination defines the criteria for the strategy to stop. For example, the strategy will stop when the goal is reached, or stop when there are no problems. There are other rules, such as maximum run time, to terminate the run.

9.5 Strategy Driven Planning Flow Chart

FIGURE 87 illustrates the sequence of steps that are followed by the rules that define strategy.

FIGURE 87 SDP Flow Chart

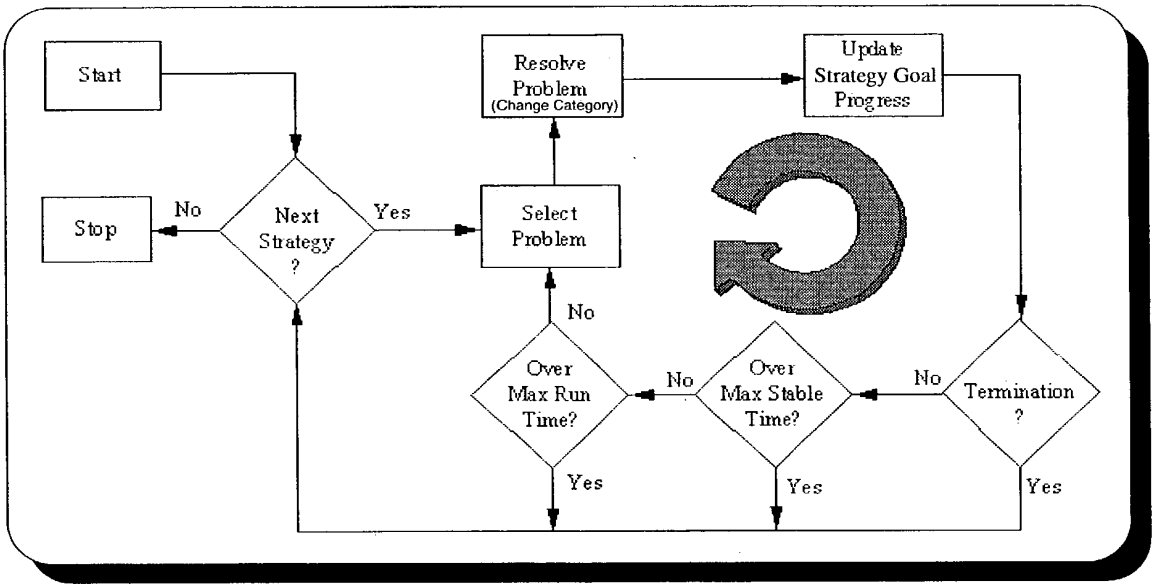


Table 20: SDP Flow Chart Explanation

Step	Action	Comments
1	Select a Strategy	If first strategy, or next strategy, then continue. Else, stop SDP.
2	Select the Problem	Select the problem set to be addressed by this strategy.
3	Resolve the Problem	Select a change category to define how to resolve the problem.
4	Update the Strategy Goal Progress	Update the goal and target to be achieved by the strategy.
5	Termination	Stop the strategy when specific criteria are satisfied. Examples include: * Max stable time is reached. * Max run time is reached.
6	Return to Step 1	

9.6 Problem-Oriented Planning

9.6.1 Description

Rhythm®'s global supply chain manager allows for Problem-Oriented Planning. This planning mechanism provides users the ability to define their own strategies for planning, decide their own goodness criteria, and how much auto-resolution they want. The system totally integrates material and capacity planning, master and operational planning, as well as factory and distribution planning. The automated planning cooperates with any manual intervention desired by the planner. The planning mechanism provides *true what-if* planning with *undo* capability.

Strategies allow users to input rules and policies to solve problems created by constraints in their system. The algorithm used is an iterative repair (or simulated annealing) type of algorithm. Global goodness criteria drive the planning problem to an optimal solution.

9.6.2 Problem-Oriented Planning Using SDP Versus Optimization

Problem-Oriented Planning using Strategy Driven Planning (SDP) allows for the intermingling of the planner's experience, knowledge, and discretion which yields a more optimal solution than the straight forward optimization approach.

SDP, being a problem oriented solution, allows the user to define the ideal plan directly and then highlight constraints that make it infeasible. Then a combination of automated, semi-automated, and manual techniques is used to intelligently change the problem constraints and solution space and build a good feasible plan.

Optimization is constrained by formulation. It requires the user to define the objective function subject to certain constraints, and forces the user to find the best plan within the constraints.

Table 21: SDP Versus Optimization User Issues

SDP User Issues	Optimization User Issues
Average user has visibility on dynamics of Supply Chain.	Supply Chain dynamics can only be understood by highly technical people with a thorough understanding of mathematics as well as business issues.
User can change constraints while solving the problem.	User cannot change constraints during the search.
User can relate to optimization trade-offs directly.	User must reverse engineer trade-offs from shadow prices.

9.6.3 Procedure

Follow the steps in Table 22 to perform Problem-Oriented Planning:

Table 22: Problem-Oriented Planning Guide

Step	Action	Comments
1	Identify the Problem	Problems are identified as one of the following categories: • Violations of policies - feasibility, desirability • Model Inaccuracies - description, dates, feasible, cost, category extensions
2	Edit the Plan to solve Problems	The plan should not be changed if no problem is solved. Policies contain localized techniques to address problems. The scope of plan adjustments should be limited.
3	Evaluate the quality of the Plan	Reducing everything to a common unit of measure does not solve a planning problem. Plans should be judged on feasibility and business sense.
4	Catalog the Problems	Problems are cataloged according to the following categories: • type • severity tolerances • interactions
5	Permit temporary Infeasibility	This opens more paths to good solutions, simplifies manual plan editing, and permits a wider range of what-if analysis.
6	Move to Feasible and Desirable	Start with a desirable, but probably infeasible plan, e.g. backward propagation of material constraints, followed by forward propagation using feasible procurements (plus capacity allocation along the way). Make search moves that improve desirability. Permit temporary infeasibility.

9.7 Strategies

A strategy can include several sub-strategies. The sub-strategies of a strategy form a sequence of strategies that are executed in order to perform that strategy. There is no inheritance of any properties between strategies and sub-strategies. A strategy with sub-strategies executes each of its sub-strategies in order and then executes itself.

Consider a strategy S which has no problem sets of its own but several sub-strategies (S1, S2,...) each of which has a problem set defined. The *problem_count* (Active_Goal model) for S in the *Plan* report shows 0. At the same time, the *problem_count*'s for strategies S1 and S2 show non-zero quantities. The *problem_count* for S should be 0, not the count of all the problems seen by its sub-strategies (i.e. sum total of S1, S2,...).

Invoking this strategy S runs its sub-strategies S1, S2,..., but the final step, running S, does nothing.

If a strategy has a *default_change_focus* (Strategy model) of 0, and no changes have been explicitly allowed, the strategy has no effect on the plan.

If a strategy has a *default_change_focus* of 100 (default) and no changes have been explicitly disallowed (i.e. none of the changes has been disallowed), the strategy will allow all of the changes with equal likelihood.

There is no difference between the following:

- strategy with its own problem set and changes and n sub-strategies
- strategy with n+1 sub-strategies, the last of which consists of the aforementioned problem sets and changes

If a strategy has both sub-strategies, and its own problem sets, the problems are solved in the following order:

- sub-strategies are performed in sequence
- strategy is performed

If a strategy has a run time of 10 minutes and it has 5 sub-strategies each of which has a run time of 10 minutes, then the run time is:

*5 * 10 minutes + 10 minutes = 60 minutes*

If a strategy is defined as Eliminate Overloads with a problem set of OVERLOAD, one sub-strategy for each resource, and each sub-strategy has a problem set of OVERLOAD and a *resource_plan_filter* of:

#.resource == Resource i

Then the behavior would not be the same as declaring no problem sets for the parent strategy. The parent strategy will focus on all OVERLOAD problems, regardless of the resource on which they occur. If no problem sets are declared, the parent strategy does nothing after running its sub-strategies.

Consider a case in which a strategy's (S) problem set is defined as all resource problems within a certain range, and a sub-strategy's (S1) problem set is defined as all resource problems for a particular resource. When the strategy is running, the parent strategy has no effect on the behavior of its children (sub-strategies).

9.7.1 Strategy Values

A strategy's main objective is to attain the most desirable goal for the company. In designing a strategy there are several values which can be used to define the strategy in order to move towards achieving that goal. Table 23 lists these values and the consequences of strategies using the values.

Table 23: Strategy Values and Consequences

If active_strategy	AND auto_run	AND running	AND resolve	THEN...
True	True	True	True	When *resolve* is invoked, the active_strategy with *running* set to true is used.
True	False	True	True	When *resolve* is invoked, the active_strategy with *running* set to true is used.
True	True	False	True	When *resolve* is invoked, the active_strategy with *auto_run* set to true is used.

9.7.1.1 Active_Strategy

active_strategy is a submodel of the Plan model. A strategy can be defined as an *active_strategy* which allows that strategy to remain active and to be kept up-to-date on every plan change. An *active_strategy* is a strategy that is maintaining active problem lists and goodness measures for its plan. The *active_strategy* models an approach to resolving the problems in a plan. It specifies the problems for which to attempt resolution, what modifications can be made in those attempts, and what criteria to use for judging the goodness of the plan.

9.7.1.2 Auto_Run

auto_run is a logical field of the Active_Strategy model. An active_strategy can be defined as *auto_run*. If an active_strategy has *auto_run* set to true, then it is run after each change to the owner plan. Therefore, if a change in the owner plan causes a problem, the active_strategy with *auto_run* set to true will be run in an attempt to resolve that problem. If an active_strategy has *auto_run* set to false, it remains unchanged.

Only one active_strategy for every plan can have *auto_run* set to true. Setting *auto_run* to true for an active_strategy will automatically set all other active strategies of this plan to false. The active_strategy with *auto_run* set to true cannot have its value changed to false unless another strategy's *auto_run* value is set to true.

To define an *active_strategy* as *auto_run*, follow the steps below:

Step	Action
1	Open the *Plan Editor* for the plan of interest.
2	Determine the active_strategy for which *auto_run* will be set to true.
3	Select the *Chooser* button next to the *Auto Run Strategy* label. A list of active_strategies will appear in a drop-down menu.
4	Select the name of the active_strategy for which *auto_run* is to be set to true. The chosen active_strategy now has *auto_run* set to true. Note that any active_strategy with *auto_run* previously set to true is now set to false.

9.7.1.3 Running

running is a logical field of the Active_Strategy model. An active_strategy can be defined as *running*. A *running* strategy is an active_strategy that is currently resolving problems in order to achieve its goal before terminating. If an active_strategy has *running* is set to true, then this strategy is currently running on the owner plan. Setting *running* to false returns the active_strategy to its initial state.

9.7.1.4 Resolve

resolve is a logical field of the Problem model. Once the active_strategy has been defined, the *resolve* function can be used to resolve any problems within the plan. The *resolve* function of the Problem model operates under the direction of the current active_strategy (with *running* set to true) in the plan. If no active_strategy has *running* set to true, then the active_strategy with *auto_run* set to true is used.

resolve is invoked by selected the *Resolve* button next to any of the problems. If *resolve* is invoked from the *Active_Strategy Editor*, the focus values from the strategy being viewed are used to resolve the problem. If *resolve* is invoked from the *Plan Editor*, all focus values are given equal weight during the resolution of the problem. See page 13 for more information of focus values.

9.7.1.5 Fields and Description

Table 24 lists the model fields relevant to defining and using strategies in Strategy Driven Planning. It also provides a description of each of the fields and the model in which the field belongs.

Table 24: Model Fields and Descriptions

Field	Description	Model
active_strategy	*active_strategy* is a submodel of the Plan model. The *active_strategy* models an approach to resolving the problems in a plan.	Plan
auto_run	*auto_run* is a logical field of the Active_Strategy model. If a change in the owner plan causes a problem, the active_strategy with *auto_run* set to true will be run in an attempt to resolve that problem.	Active_Strategy
running	*running* is a logical field of the Active_Strategy model. If an active_strategy has *running* is set to true, then this strategy is currently running on the owner plan.	Active_Strategy
resolve	*resolve* is a logical field of the Problem model. Once the active_strategy has been defined, the *resolve* function can be used to resolve any problems within the plan.	Problem

9.8 Strategy Goodness Measurement

9.8.1 Description

The strategy goodness measurement consists of a parameterized goodness equation with adjustable weights that are maintained by strategies and updated as problems are created and resolved.

9.8.2 Procedure

The basic steps to follow for resolving problems by using strategies are shown in Table 25:

Table 25: Resolving Problems

Step	Action
1	Select problem
2	Solve problem
3	Update objective function

9.8.3 Relevant Models and Fields

Some particular fields within some models are specific to the strategy goodness measurement. See Table 26.

Table 26: Goodness Fields

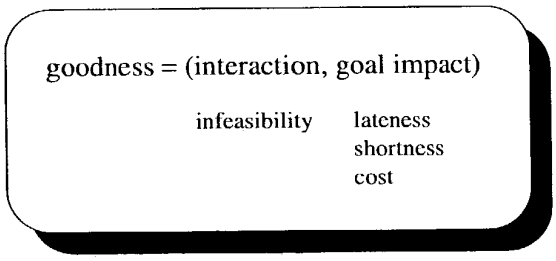
Model	Field
Active_Goal	goal
Active_Strategy	interaction
	annealing_goodness
Problem	interaction
	resolve
Strategy	max_run_time
	max_stable_time
	termination

9.8.4 Definitions

9.8.4.1 Goodness

Goodness is the feasible impact on the goal. It is a function of *interaction* and the *goal*'s target value. *annealing_goodness* is an export-only field in the Active_Strategy model which returns a number representing the comparative goodness of the plan. Initially, the *annealing_goodness* will return a value of 1. Smaller goodness indicates a better plan, judged by its goals.

The goodness function is used to determine if the plan is getting better or worse.



9.8.4.2 Interaction

Interaction measures the impact of infeasibility on the goal. It is an estimate of how bad the problem is: how much work (how many plan changes) will be needed to resolve the problem. *interaction* is an export-only field in the Problem model which returns the sum of interaction values from all active problems.

The Strategy relies on the interaction. With respect to lateness, the unit of measure for interaction would be time. With respect to shortness, the unit of measure for interaction would be quantity.

High interaction is not good or bad. It just exists as such. The strategy tends to prefer selecting higher interaction problems to resolve before lower interaction problems.

At the end, the *interaction* value should be as close to 0 as possible, so it should be penalized as much as possible.

9.8.4.3 Using Multiple Goals

Two goals can be used. For example, if no (or minimal) shortness and no (or minimal) lateness) are desired, then the *goal* field in the Active_Goal model should indicate both MINIMIZE_SHORTNESS and MINIMIZE_LATENESS.

9.8.4.4 Focus

9.8.4.4.1 Description

Focus numbers are used as follows:

Suppose for a given strategy, a MOVE_IN of 75 and MOVE_OUT of 25 are defined. These focus numbers stay the same, but they are used by resolvers along with the current state of the plan. For instance, the resource problem resolvers choose whether to move in, out, or off. They prefer moves to open space closest to the current plan. If open space before the start of the current operation plan is two hours away, but open space after the start of the current operation plan is two weeks away, the resolvers weight more heavily the move in over the move out.

9.8.4.4.2 Example

The parameters provide weights and the situation further weights the decision. In the following example, start by resolving resource problems because, although they have a lower focus value, they have more calculated problems (0.5 * 150 = 75). So, select the interaction (strategy) based on the resource problems.

	focus	#problems			
Buffer Problems	1.0	50	=	50	
Resource Problems	0.5	150	=	75	
				125	Total Problems

9.8.5 Measuring Goodness

9.8.5.1 Description

If Supply Chain Planner (SCP) kept within feasible space, goodness would be a relatively simple, programmable cost function on late orders, early orders, short orders, resource expenses, etc. But SCP does its work in infeasible space, so goodness needs to measure infeasible plans.

9.8.5.2 Definitions

9.8.5.2.1 Feasible Space Goodness

Feasible space goodness (FSG) is a function of feasible problems such as order fulfillment and resource costs. It does not cover infeasible problems such as illegally overlapping operations.

9.8.5.2.2 Infeasible Space Goodness

Infeasible space goodness (ISG) is a function which estimates what FSG would return if we performed some set of changes C to get to feasible space. ISG is a function of feasible and infeasible problems. It includes FSG plus estimates of the impact on FSG of resolving infeasible problems such as illegally overlapping operations.

9.8.5.2.3 Pain

Each infeasible problem estimates its *pain* (the likely number of problems that will be incurred to eliminate that infeasibility). ISG is the sum of the pain. We want to drive pain down. The pain function is a quick and dirty way of estimating which one causes the least pain. For example, if we have three load plans, determine (guess) which load plan causes the least pain, as measured by e.g. 100 units of pain, when moved in or moved out.

9.8.5.3 Annealing Goodness

FSG is the goodness measure that is user specified as *annealing_goodness* in the Active_Strategy model. The goal is to minimize FSG, but zero out the list of infeasibilities. That is, which infeasibilities to drive out is a separate user specification in the Strategy from what the global goodness measure is. An infeasibility has infinite badness in the users' final global goodness (so is not added).

9.8.6 Example of Changes

An example of changes C is forward simulation under a dispatch rule such as First-Come-First-Serve (FCFS). So, if illegal operation overlap occurs at time T, the impact on FSG components such as due dates from resolution of the overlap via forward simulation is estimated. In an overload of three operations, the first arriving operation would not be pushed. The other two would. This is not to say that the simulation is performed. The first operation is skipped in assessing this problem's severity. (Note that if it is late anyway, that would show up in downstream problems, accounted for in their costs.)

Another example of changes C is cancellation. On a resource overload, the impact on due dates, etc., is estimated from cancelling operations such that the overload would be resolved.

But a better change C might be an estimate of the most likely set of changes SCP will make, based on change pain functions. For instance, if the overload can be resolved easily by moving to an alternate which has plenty of capacity, the pain is zero, so SCP's die roll will likely get that change. So, this particular overload problem has small severity. Other overload problems without the luxury of unloaded alternate resources have higher severity. So, they will be resolved first. This attacks the current-plan bottlenecks first.

9.8.7 Estimating the Impact of Infeasible Problems

The following rules apply to estimating the impact of infeasible problems:

- The problem works downstream in the form of buffer arrival delays, impacting increasing numbers of orders. The problem spreads out as it goes downstream.
- While working downstream, the orders to choose to push or *move to alternate*, etc., should depend on the given FSG parameters. For instance, if the customer sets his FSG parameters such that a late big order is preferable to a lot of late small orders, then you want to tend to push a big order instead of a lot of small orders.
- While Strategy specifies which changes are permitted, the specific problem's *resolve()* function chooses among them. So we cannot rely on change pain estimates to determine change C unless *resolve()* provides information further weighting the various available changes.

9.8.8 Terminating a Strategy

The strategy may be terminated by setting one of the following fields in the Strategy model:

max_stable_time - the Strategy will terminate if it does not find an improvement to the plan within *max_stable_time*.

max_run_time - the Strategy will terminate if it runs for longer than *max_run_time*.

When the strategy terminates, it will return to the best plan found so far.

9.9 Strategy Construction

A strategy can be created to meet a business's goal, whose problem set covers every feasible problem, and whose change_categories are completely unrestricted and equally weighted. This strategy would, in theory, return a feasible plan which meets the goal. However, this strategy is not utilizing the advantages of Strategy Driven Planning.

To maximize the potential of Strategy Driven Planning, some basic building principles should be considered:

- Make sure the goal is attainable. Do not make the goal over-specific.
- Break the whole into pieces and focus on the pieces.
- Narrow the change_categories and change their weights, thus focusing SDP on more productive problem-solving.
- Push the plan towards meeting the goal but have infeasibility where the plan can be maneuvered manually toward feasibility.

Using Strategy Driven Planning in this manner will provide much more business value.

Table 27 describes the basic building principles of Strategy Driven Planning.

Table 27: Steps for Strategy Construction

Step	Description	Comments
1	Define the Goal	Define the goal without being too specific. The goal will be more attainable with fewer parameters.
2	Focus on the Pieces	Attack the plan in pieces. Solving the problems of the pieces will lead to solving the problems of the whole.
3	Narrow the Change_Categories	Make the change_categories more specific and change their weights. This helps SDP to target the problems and to be more productive.
4	Temporary Infeasibility can be Good	Remember that infeasibility is okay temporarily and can help the plan move toward feasibility more quickly.

9.10 Case Study of Strategy Driven Planning

9.10.1 Introduction

The following example illustrates one application of strategy driven planning.

9.10.2 Assumptions

Customer A serves its Market1 and uses Market2 opportunities to fill unused capacity. This is mainly due to significantly larger margins on Market1 sales.

9.10.3 Feasible Solution for Market1 Commitments

The procedure:

- Create two sellers, Seller1 and Seller2
- Read in two forecasts, for Seller1 and Seller2
- Satisfy all of Seller1 - when all requests are satisfied, the system generates plans to satisfy the commitments from sales
- Resolve (either manually or otherwise) whatever problems may have arisen. At this point, a feasible solution should exist for all commitments.
- Once a plan exists for the Market1 market, mark it released
- Tell the strategy not to move or resize the released plans

9.10.4 Feasible Solution for Market2 Requests

If all Market2 requests are satisfied after the feasible solution is obtained for all Market1 commitments, the system tries to satisfy all Market2 requests, creating problems wherever they occur. At this point, a strategy should be invoked which solves the resulting problems without affecting allocations for Market1 production. The following strategy approaches exist for solving the resulting problems:

- Require that this strategy be entirely manual
- An alternative approach would be to:
  - Use some means to distinguish Market1 and Market2 requests
  - Go through the steps outlined above
  - Run a strategy which has the following problem sets:
    - One consisting of REQUEST and PROMISE problems for domestic requests with a high focus
    - One or more problem sets which focus on all resource and buffer problems but with a lower focus

  This restricts the impact of problem solving to Market2 orders, keeping Market1 allocations at existing levels. Assuming this converges to a feasible plan, the plan should have left Market1 allocations untouched while scaling back Market2 allocations to the extent necessary to achieve feasibility.

9.11 Buffer Problem Resolvers

9.11.1 Introduction

A buffer problem resolver is the action taken to resolve a buffer material problem. Rhythm uses these resolvers when planning to satisfy a request. This section describes how buffer problem resolvers work. It lists the types of buffer problems, and then describes the various ways in which each type of problem can be resolved.

9.11.2 Types of Buffer Problems

There are three main types of buffer problems:

- NEGATIVE_ON_HAND - arises anytime the amount of material in a buffer drops below zero.
- LOW_ON_HAND - created when there is still some material in the buffer, but less than the safety level, or *min_on_hand*, for that buffer.
- EXCESS_ON_HAND - occurs when a buffer has more material on hand than is needed. Some buffers can define (through their flow policies) an *excess_on_hand* field that specifies a threshold value. The problem appears only if the amount of excess material is greater than that threshold.

9.11.3 Resolving Buffer Problems

Rhythm resolves buffer problems in different ways, depending on two factors: the buffer flow policy and the *change_category* set by the strategy. For example, the strategy can be defined to focus most heavily on buffer problems, moderately on resource problems, and lightly on promise plan problems. Strategy change categories used by buffer problem resolvers are described in this section.

One flow policy may be defined to model the flows to and from a buffer, but not to detect problems or create replenishment operations. Another flow policy might specify that Rhythm issue a supplying operation with the same quantity each time an operation consumes from a buffer. See the *Rhythm Model Reference Manual*, the *Buffer Extensions* section for a complete description of buffer flow policies.

9.11.3.1 NEGATIVE_ON_HAND and LOW_ON_HAND

Both NEGATIVE_ON_HAND and LOW_ON_HAND problems are resolved the same way. (NEGATIVE_ON_HAND can be thought of as LOW_ON_HAND with a safety stock level of zero units.)

NEGATIVE_ON_HAND and LOW_ON_HAND problems are resolved in one of the ways shown in Table 28, by manipulating supplying or consuming flow plans. The resolver selected depends on the buffer flow policy and the *change_category* values set by the strategy. If the strategy focus is the *change_category* listed in Table 28, then Rhythm takes the action listed in the *Resolver Selected* column.

Table 28: NEGATIVE_ON_HAND and LOW_ON_HAND Resolvers

Resolver Selected	Strategy *change_category*
Increase the supply quantity of a supplier	RESIZE
Decrease the consumed quantity of a consumer	RESIZE
Move a supplier to an earlier date	MOVE_IN
Move a consumer to a later date	MOVE_OUT
Add a new supplier	USE_MORE

9.11.3.2 Selecting a Resolver for NEGATIVE_ON_HAND and LOW_ON_HAND

In addition to the strategy focus items listed in Table 28, other issues taken into account when deciding which resolver to use are determined by the buffer flow policy. Flow policies and their impact on NEGATIVE_ON_HAND and LOW_ON_HAND problem resolvers are described next.

- INFINITE - The simplest of all flow policies. There is no resolver. In fact, NEGATIVE_ON_HAND and LOW_ON_HAND problems never appear on INFINITE buffers.
- FIXED_QUANTITY / FIXED_QUANTITY_FENCED - These flow policies never resolve a NEGATIVE_ON_HAND by resizing a supplying flow plan, since all supplying flow plans must be kept at the specified quantity. They try to resolve, if necessary, by:
  - moving in a supplier
  - moving out or resizing a consumer
  - creating a new supplier
- ON_HAND_CALENDAR and SUPPLY_CALENDAR- All of the supply in an ON_HAND_CALENDAR or a SUPPLY_CALENDAR buffer is presumed to come from the calendar allotment, so they can resolve a NEGATIVE_ON_HAND by trying to move a consuming flow plan in or out to a date when the calendar still shows material available, or by resizing a flow plan. It never resolves by creating a supplier.

9.11.3.3 Example

Suppose you have a buffer with the flow_policy of ON_HAND_CALENDAR. Assume the amount on hand according to the calendar is 100 a day up to the 12th day of the month, 50 a day from the 12th to the 15th day of the month, and 0 a day thereafter. The operation plan created is for 60 and pulls from this buffer on the 13th, therefore it is pulling 10 more than is available. Take the following steps to resolve the NEGATIVE_ON_HAND problem created:

- Open the *Site Plan Editor.*
- Select the *Problems* tab and check for NEGATIVE_ON_HAND problems.
- Click the *Resolve* button to see a NEGATIVE_ON_HAND problem.

Resolving the NEGATIVE_ON_HAND results in the operation plan performing a MOVE_IN to an earlier data, when there is more available on hand.

9.11.3.4 EXCESS_ON_HAND

EXCESS_ON_HAND problems are resolved in one of the ways shown in Table 29. The resolvers involve manipulating supplying or consuming flow plans, basically the opposites of the five NEGATIVE_ON_HAND resolvers. The resolver selected depends on the flow policy and the *change_category* that is focused by the strategy.

Table 29: EXCESS_ON_HAND Resolvers

Resolver Selected	Strategy *change_category*
Decrease or delete the supply quantity of a supplier	RESIZE
Increase the consumed quantity of a consumer	RESIZE
Move a consumer to an earlier date	MOVE_IN
Move a supplier to a later date	MOVE_OUT
Add a new consumer	USE_MORE

9.11.3.5 Selecting a Resolver for EXCESS_ON_HAND

In addition to the strategy focus items listed in Table 29, other issues taken into account when deciding which resolver to use are determined by the buffer flow policy. Flow policies and their impact on EXCESS_ON_HAND problem resolvers are described next.

- INFINITE - In a sense, there is always EXCESS_ON_HAND in an INFINITE buffer. It is never flagged as a problem.

- FIXED_QUANTITY and FIXED_QUANTITY_FENCED - These flow policies cannot resize their supplying operations, so they can only resolve by:
  - increasing a consumer
  - moving in or out
  - deleting whole suppliers, if the excess is larger than the supplying operation quantity
- ON_HAND_CALENDAR and SUPPLY_CALENDAR- Since the material in an ON_HAND_CALENDAR or a SUPPLY_CALENDAR buffer is thought of more as capacity than as actual material, EXCESS_ON_HAND is never detected in these buffers.

☞ Note: Rhythm never makes changes to released operations. Only those operations not yet released are modified by buffer problem resolvers.

9.11.4 Using Alternate Operations to Resolve Buffer Problems

An additional strategy *change_category* that can affect buffer problem resolution is USE_ALTERNATE_OPERATION. As with other strategy focus change categories, the focus can vary. If set to zero, Rhythm does not use alternate operations. Although changing to an alternate operation cannot always help solve buffer problems, there are situations where it can. An example is transportation, where moving from ground to air transportation may solve a problem related to receiving materials when needed.

9.12 LFL Buffers

9.12.1 Description

References to *LFL Buffers* mean any buffer with a Flow_Policy of LFL_SIMPLE, LFL_BOUNDED, or MLFL_BOUNDED. (LFL = Lot-for-Lot, MLFL = Many-Lots-for-Lot.)

9.12.2 Problem Categories

To identify problems in a way that is more appropriate to the hard pegging done in LFL Buffers, there are two problem categories, LOT_OVER_SUPPLIED and LOT_UNDER_SUPPLIED (category extensions of the Problem model). These categories correspond roughly to EXCESS_ON_HAND and NEGATIVE_ON_HAND, but with a few key differences:

Each LOT_OVER_SUPPLIED or LOT_UNDER_SUPPLIED problem is associated with a specific pegging of suppliers and consumers in the Buffer. They are not associated with the total on_hand value of the buffer being high or low at any particular time, the way EXCESS_ON_HAND and NEGATIVE_ON_HAND are.

9.12.3 Example

For example, consider a Buffer_Plan with the following Flow_Plans:

Date	*Quantity*	*Pegging*	*On_Hand*
d0	*20*	*A*	*20*
d1	*10*	*B*	*30*
d2	*-10*	*A*	*20*
d3	*-20*	*B*	*0*

Notice that there is a difficulty with both peggings. A supplies 20 units to a 10 unit consumer, and B supplies only 10 to a 20 unit consumer.

Two problems are raised:

- LOT_OVER_SUPPLIED problem for A (from d0 to d2)
- LOT_UNDER_SUPPLIED problem for B (from d1 to d3)

LOT_UNDER_SUPPLIED is a feasibility problem; LOT_OVER_SUPPLIED is not.

Like NEGATIVE_ON_HAND and EXCESS_ON_HAND, LOT_OVER_SUPPLIED and LOT_UNDER_SUPPLIED can be caused by time issues as well as quantity issues. For example, if a 10 unit consumer is pegged to a 10 unit supplier, there will still be a LOT_UNDER_SUPPLIED problem if the consumer starts before the supplier ends, or a LOT_OVER_SUPPLIED problem if the supplier ends before the consumer starts.

9.12.4 Problems Handled Automatically

Rhythm does not automatically handle any NEGATIVE_ON_HAND that stretches to infinite future, as do other Flow_Policies. Instead, it automatically handles the problem created when a new Flow_Plan consumes from the Buffer_Plan, and does not have a supplying Flow_Plan (or Flow_Plans, in the case of MLFL_BOUNDED) pegged to it. In that case, the Buffer creates an upstream supplying operation and pegs its supplying Flow_Plan to the new consumer.

This means that you could see the on_hand going negative out to infinite future, but it will still have created upstream operations for the demand.

9.12.5 LFL Resolvers

The resolvers for LOT_OVER_SUPPLIED and LOT_UNDER_SUPPLIED are similar to those for NEGATIVE_ON_HAND and EXCESS_ON_HAND, but generally work a little smarter, because they consider only their pegging, not the entire Buffer_Plan. Here are the possible ways they can resolve:

Table 30: LOT_UNDER_SUPPLIED Resolvers

Resolver	Change Category	Notes
move in a supplier	MOVE_IN	Only if there is a supplier pegged later than a consumer.
move out a consumer	MOVE_OUT	<same>
increase a supplier	RESIZE	Only if there is a small supplier pegged to a large consumer.
decrease a consumer	RESIZE	<same>
create a new supplier	USE_MORE	Only in MLFL_BOUNDED, only if the consumer is larger than all the suppliers, and only if the problem cannot be solved by resizing.

Table 31: LOT_OVER_SUPPLIED Resolvers

Resolver	Change Category	Notes
move in a consumer	MOVE_IN	Only if there is a consumer pegged later than a supplier.
move out a supplier	MOVE_OUT	<same>

Table 31: LOT_OVER_SUPPLIED Resolvers

Resolver	Change Category	Notes
increase a consumer	RESIZE	Only if there is a small consumer pegged to a large supplier.
decrease a supplier	RESIZE	<same>
create a new consumer	USE_MORE	Only if the consuming operation field is set, and only if there is a supplier without a consumer already pegged to it.

If the *consuming_operation* field is not set, and a supplier is created, a LOT_OVER_SUPPLIED problem is raised for that supplier. That problem will not be resolved until the *consuming_operation* field is set, or until a new consumer is created (in which case, it will be pegged to the free supplier).

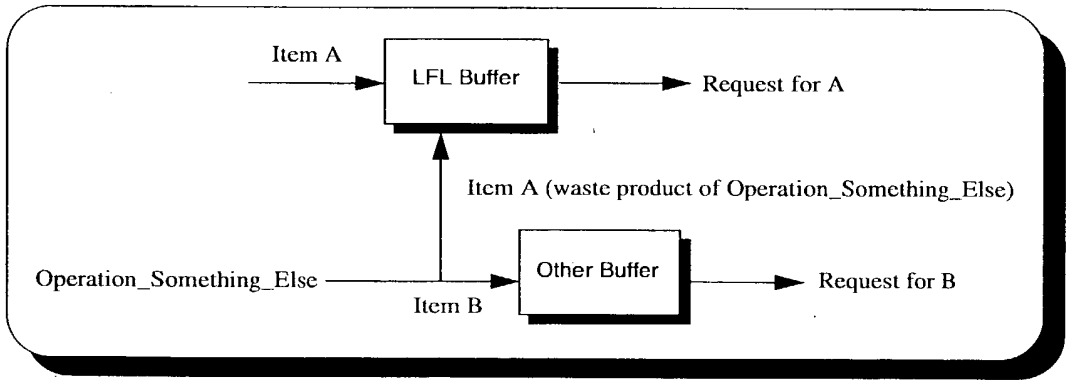
If a problem of LOT_OVER_SUPPLIED exists, it means that the consuming operation requires less than the supplying operation.

9.12.6 Example 1

If the supplier is released, then there is no option to resize it, that is how much is coming. It can only be consumed.

In FIGURE 88, the LFL Buffer holds Item A. Suppose that Operation_Something_Else produces Item B, but Item A is a by-product or a waste. It has to go somewhere, so it is directed into the LFL Buffer. The resolution for the LOT_OVER_SUPPLIED problem is to create a corresponding downstream delivery. You cannot choose to not accept the waste Item A.

FIGURE 88 LOT_OVER_SUPPLIED



If the user explicitly resizes the supplier, it would be rude to resize it back down. The user would almost surely prefer you to resize the consumer accordingly.

9.12.7 Example 2

LOT_OVER_SUPPLIED problems might also be caused by a time difference between the supplying and consuming operations. For example, a period of zero efficiency at the Resource loaded by the Operation assembly can result in the supplying operations plans not being able to MOVE_IN. Therefore, the only way to *Resolve* these problems is for the consuming operation to be moved in, thus creating a PROMISE_PLANNED_EARLY problem. To perform this, set a *hint* such that the consuming operations start before the supplying operations end, a LOT_UNDER_SUPPLIED problem is created, which can be resolved using the *Resolver*.

9.13 Resource Problems

The following resource problems exist:

- Overlap problems - simpler than oversize problems because size capacity data does not need to be set up.
- Overlap problems - created whenever a resource with EXCLUSIVE_USE_LOAD_POLICY has load plans which overlap

Resource problems occur as a result of the following actions. First, the load policy of a resource that gets populated with load plans is set to EXCLUSIVE_USE. This can be done in the data, or by editing the user interface. Next, the load plans are arranged in one of the following ways such that they overlap:

- On a shipping resource which is the last resource visited in the assembly tree, setting two request dates equal to each other generates an initial plan having the two Shipping load plans over the same period. A resource problem results.
- Another action that generates resource problems is to load enough requests such that overlap occurs.
- Another action that generates resource problems is to edit load plan dates such that they overlap.

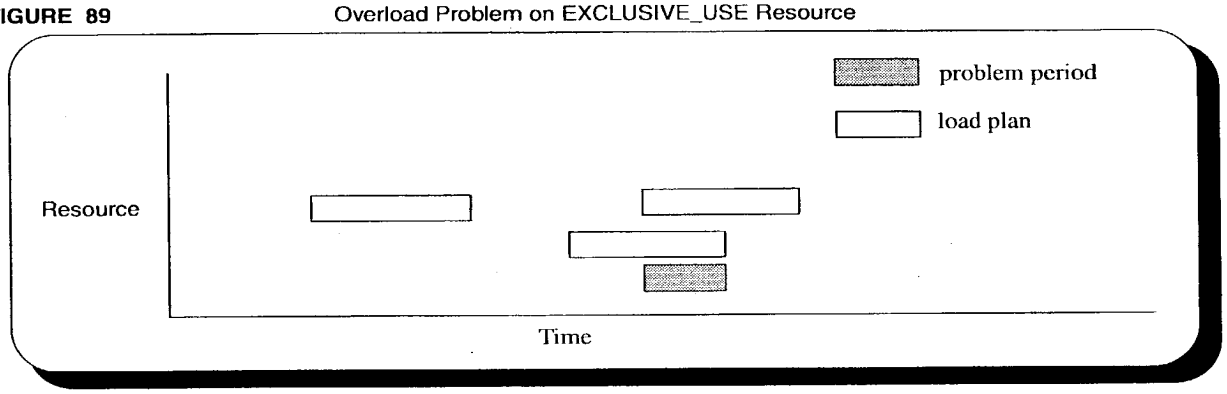
9.14 Resource Balancing

This section describes how the OVERLOAD and OVERSIZE problem resolvers work. These two resolvers effectively cause resources to balance.

9.14.1 Overload

OVERLOAD problems occur on EXCLUSIVE_USE resources whenever more than one load plan runs at a given time. OVERLOAD can be viewed as an OVERSIZE problem where each load plan size is 1 and the resource's maximum count size is 1. Note the problem period in FIGURE 89:

FIGURE 89 Overload Problem on EXCLUSIVE_USE Resource

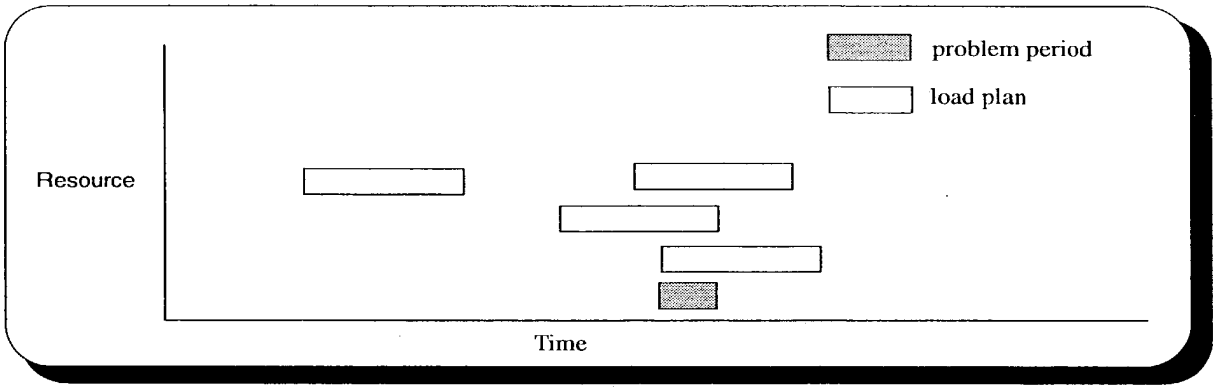


9.14.2 Oversize

OVERSIZE problems occur on SHARED_USE resources whenever the load plans at a given time add up to more size than is available. OVERLOAD can be viewed as an OVERSIZE problem where each load plan size is 1 and the resource's maximum count size is 1. The description of OVERSIZE resolver should be applied to OVERLOAD with this fact in mind.

Size is measured in one of several ways. See the Size models. One is maximum count which is just the count of load plans at any given time. If maximum count is 2, then FIGURE 90 shows OVERSIZE problems. Note that the problem period occurs where three load plans overlap:

FIGURE 90 Oversize Problem on SHARED_USE Resource



9.14.3 Big Problem Resolver

The big problem resolver works as follows:

- estimates the number of load plans needed to be moved out of the problem period. The estimation is as follows:

  excess load plans = num_load_plans - (num_load_plans / load_percentage)

- divides this number into move_ins, move_outs, and move_offs proportional to strategy focus numbers.
- iterates, selecting earliest load plan remaining in the problem period and moving in, out, or off. Since some or all load plans may not have an alternate resource, the actual number of move_offs may vary from this; the difference is made up by extra move_outs.

When moving in, load plans move to a gap where available size minus used size is no smaller than the load plan's used size. Such a gap's period can be smaller than required to accommodate the entire load plan however. Thus at times the problem is shifted rather than solved. The general behavior desired is to spread out a big problem, usually reducing it into several smaller problems. Moving out involves similar logic.

When moving off, the start and end dates are not restricted, so load plans will tend to stay at their current dates. This will usually cause problems at the alternate. If these problems are big, this logic is reapplied on the alternate, but on fewer load plans assuming move_in focus or move_out focus was nonzero.

9.14.4 Small Problem Resolver

The small problem resolver works as follows:

- iterates, trying to move load plans out of the problem period, capping iterations at the original number of load plans. If a move causes the problem's load percentage to reach 100% or lower, the loop is terminated. The load percentage is the load plans' standard time divided by capacity, accounting for both efficiency and size, (e.g. see the *sized_capacity* field of model Resource_Plan).

  While more precise than the big problem resolver, even in the termination condition, no attempt is made to completely eliminate the problem. For instance, suppose capacity in the problem period is 10 hours. After several moves, there may be 4 load plans totalling 10 hours. The loop is terminated. But, the problem is not resolved if the maximum size is 3 and all 4 load plans overlap a given time. Rather, the problem is reduced. Such imprecision allows strategy to give other problem resolvers a turn and to return to solve the rest of this problem later.

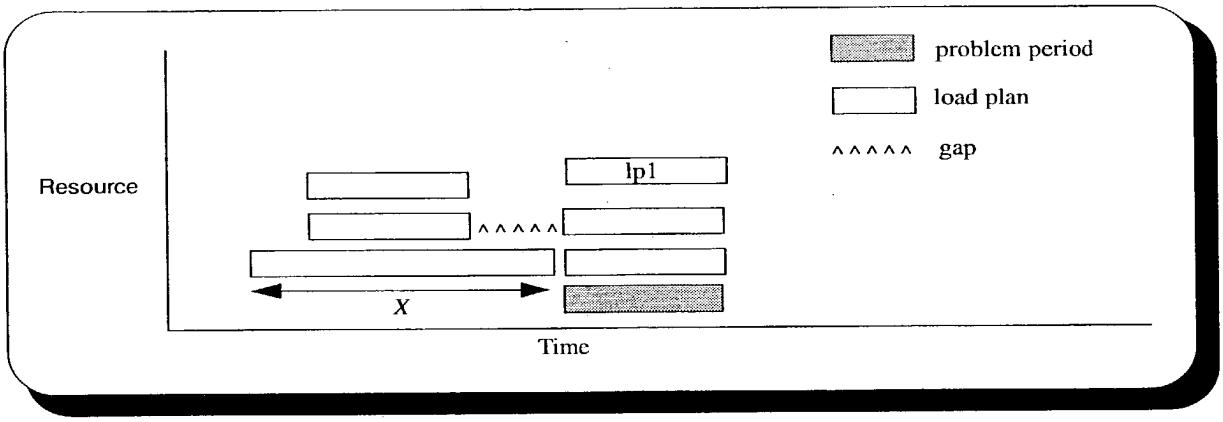
- Within each iteration the following occurs:
  - Select N random load plans.
  - Determine the least distance to open space for each load plan, as described later. 90% of the time, the load plan with the least distance is moved; for generality, 10% of the time a random one is moved.

  Least distance to open space is:

  min(distance_to_earlier_open_space,
  distance_to_later_open_space,
  distance_to_alt_open_space)

  distance_to_earlier_open_space is the time searching backwards from the problem start to the point where gaps (as described under the big problem resolver) have capacity sufficient to accommodate the load plan's standard time. For example: efficiency is fixed at 50%. Size maximum count is 2, so only two load plans can run at a given time. FIGURE 91. Each character represents 1 hour of real time (and 0.5 hours of standard load time, due to 50% efficiency).

FIGURE 91 Small Problem



distance_to_earlier_open_space of lp1 for this problem is X. The gap closest to lp1 accommodates 5 hours (2.5 standard hours) of lp1, and the gap furthest accommodates the other 5/2.5 hours of lp1.

A date to which lp1 can be moved without causing problems is not the goal. The date is a heuristic jump closer to feasibility.

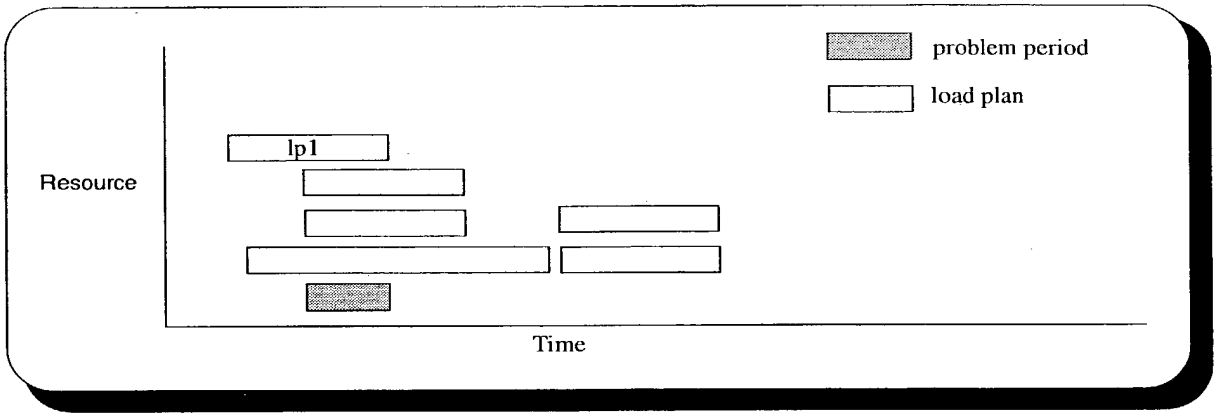
distance_to_later_open_space is defined similarly, only searching forwards from problem end.

distance_to_alt_open_space is infinite when there are no alternate resources. Otherwise it is the minimum of distances to earlier/later open spaces among the alternate resources.

These distances are weighted based on strategy focus to determine which moves to perform, and a die roll determines which is selected. The date moved to is one of the open space dates. For instance, if move_in, the load plan moves to start at the X in the above example. In the case of moving_off, the open space dates are ignored.

In the above example, lp1 moves as shown in FIGURE 92, shifting and reducing the problem:

FIGURE 92 Shift Problem



That problem would later be resolved either by moving lp1 again, or moving other load plans in that period. See FIGURE 93 and FIGURE 94. Note that there is no problem (no overlap of more than two load plans at any one time).

FIGURE 93 Resolve Problem - Method 1

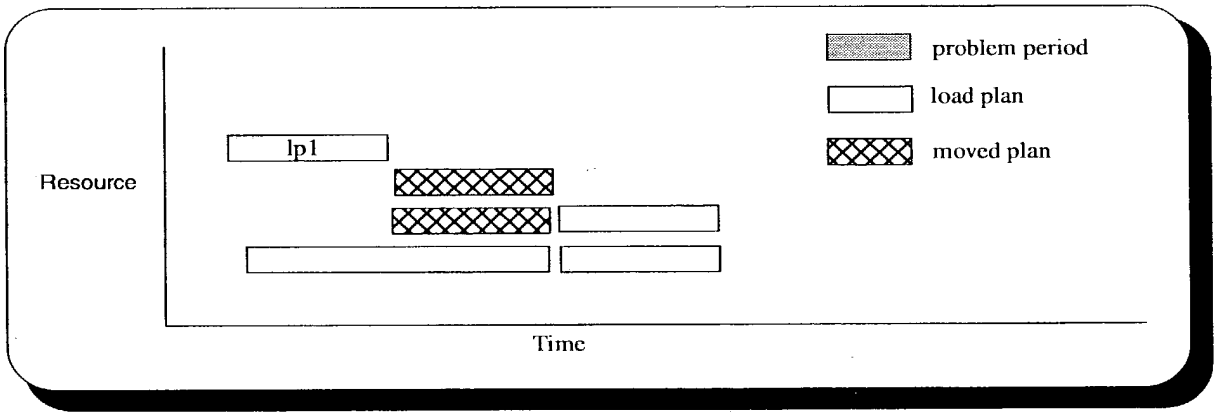
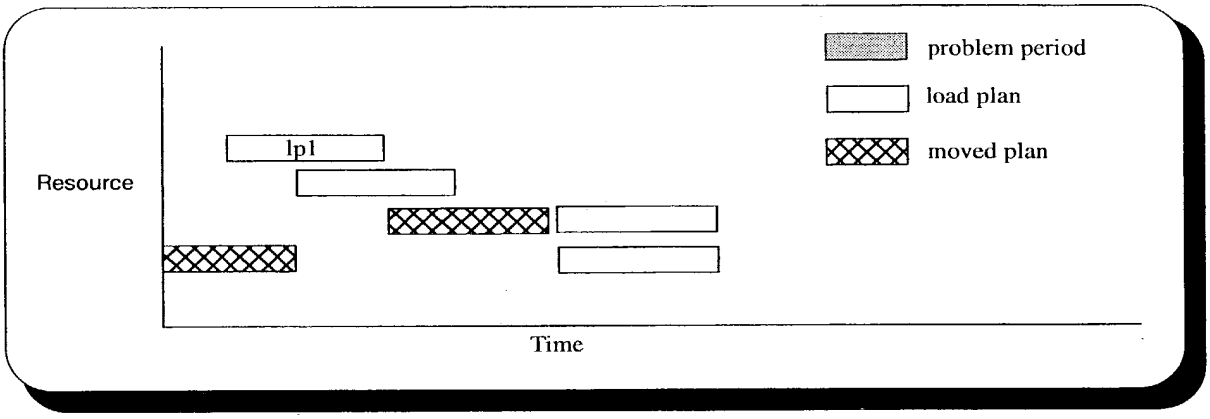


FIGURE 94 Resolve Problem - Method 2



9.15 Alternate Operations

See the *Rhythm*® SCP Standard Reports Manual.

9.16 Alternate Parts

The automated planning function has the capability to accept WIP at alternate parts. The WIP reporting from the external system contains information on operation name, quantity, and time. Rhythm takes in this information and compares it with projected WIP from the plan it generates. The result of the comparison is as follows:

- Reported WIP is less than planned WIP:

  Action options:

  - Check alternate parts
  - Start new wafers
  - Leave it short (planner could choose one of the first two)
- Reported WIP is more than planned demand:

  Action options:

  - Process the WIP and store it in the next stopping point which is either die bank or FGS
  - Other orders should be able to use up the excess

9.17 Alternate Resources

Changing to alternate resources is the setting of a Load Plan's Resource Plan. It can be set, and should be able to propagate plan changes.

Glossary

basic reports Reports such as the *Main* window and *User* report which provide starting points into each *Rhythm*® product.

buffer Models the management of the flow of interchangeable Items handles material/ inventory planning. Holds items between operations.

buffer plan Models flow of material over time. A model of the management of an Item at a particular Location (a SKU).

buffer state Initial contents of a Buffer.

capacity Defines how the capacity level of a Resource is modeled.

confirmation dialog A window that asks a Yes or No question, verifying the requested action.

constrain Any element that prevents the system from achieving the goal of making more money.

flow Items that are produced or consumed by an Operation.

flow plan Models buffer quantities over time.

flow policy Specifies how to replenish a buffer.

item A family (tree) of parts.

inventory The quantity of money invested in materials that the firm intends to sell.

load Defines how a Resource is loaded by an Operation.

load plan Represents the plan for placement of a particular Load on a Resource.

load policy Defines how the Operation Loads placed on a Resource are planned.

main report The first window displayed when you start the *Rhythm*® user interface.

operation Represents a process, activity or action that transforms Items into other Items and consumes the capacity of Resources.

operation plan The plan for a particular activity to be performed, as specified by the Operation.

operation state Records initial state of an operation.

overlapping skill calendars Skill calendars that have a non-zero efficiency for more than one skill at the same time.

problem An infeasible or undesirable condition such as illegally overlapping operations and late orders.

problem detector A mechanism for detecting a specific type of Problem. The various extensions have specific classes of problems they automatically detect. In addition to that, the user can add various problem detectors for problems not already detected by the extensions.

problem-oriented planning A *Rhythm*® planning mechanism that provides you the ability to define your own strategies for planning, decide your own goodness criteria, and how much auto resolution you want.

process Description of the process performed by an Operation, restrictions on that process, and rules on how to resolve Problems.

report name The title displayed at the top of a report.

resource Models the capacity to perform Operations.

resource plan Models the loads on a resource over time.

resource state Models the state of a Resource: setup, maintenance, queue, activity, up/down.

RhythmLink A product that automates the task of extracting data from existing databases and loading it into *Rhythm*®.

seller A channel.

skill Models a basic capability provided by Resources and needed to perform Operations.

storage Defines what happens to the Items in a Buffer while they are stored.

strategy A model that directs policy-based problem solving algorithms.

strategy-driven planning (SDP) Planning based on carefully defined strategies. See **strategy**.

throughput The total volume of production through a facility, or the quantity of money generated by a firm over a period of time.

usage policy Models usage(consumption/production) of a particular item by an Operation.

utility reports Reports such as *Delete* and *Save As* which provide some of the basic screenware functions in *Rhythm*®.

Index

The Intelligent Solution for
Global Supply Chain Management

Worldwide Headquarters
909 E. Las Colinas Blvd.
16th Floor
Irving, Texas 75039 USA
Phone 1-800-800-3288
214-860-6000
Fax 214-860-6060
E-mail info@i2.com
www.i2.com

Offices in: Atlanta · Boston · Charlotte · Chicago · Detroit · Los Angeles · New York · Pittsburgh · Santa Clara
Brussels · Copenhagen · London · Melbourne · Mexico City · Munich · Paris · Singapore · Sydney · Tokyo · Toronto

US 094155070QP1

Creation date: 08-13-2004
Indexing Officer: THSU1 - TONY HSU
Team: 3600PrintWorkingFolder
Dossier: 09415507

Legal Date: 12-15-2003

No.	Doccode	Number of pages
1	AF/D	70
2	AF/D	344

Total number of pages: 414

Remarks:

Order of re-scan issued on .....